Programming up to Congruence (Extended version)

Vilhelm Sjöberg Stephanie Weirich
University of Pennsylvania, Philadelphia, PA, USA
{vilhelm,sweirich}@cis.upenn.edu

Abstract

This paper presents the design of ZOMBIE, a dependently-typed programming language that uses an adaptation of a congruence closure algorithm for proof and type inference. This algorithm allows the type checker to automatically use equality assumptions from the context when reasoning about equality. Most dependently-typed languages automatically use equalities that follow from β-reduction during type checking; however, such reasoning is incompatible with congruence closure. In contrast, ZOMBIE does not use automatic β-reduction because types may contain potentially diverging terms. Therefore ZOMBIE provides a unique opportunity to explore an alternative definition of equivalence in dependently-typed language design.

Our work includes the specification of the language via a bidirectional type system, which works "up-to-congruence," and an algorithm for elaborating expressions in this language to an explicitly typed core language. We prove that our elaboration algorithm is complete with respect to the source type system, and always produces well typed terms in the core language. This algorithm has been implemented in the ZOMBIE language, which includes general recursion, irrelevant arguments, heterogeneous equality and datatypes.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords Dependent types; Congruence closure

1. Introduction

The ZOMBIE language [10] aims to provide a smooth path from ordinary functional programming in a language like Haskell to dependently typed programming in a language like Agda. However, one significant difference between Haskell and Agda is that in the latter, programmers must show that every function terminates. Such proofs often require delicate reasoning, especially when they must be done in conjunction with the function definition. In contrast, ZOMBIE includes arbitrary nontermination, relying on the type system to track whether an expression has been typechecked in the normalizing fragment of the language.

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676974

Prior work on ZOMBIE [10, 28] has focused on the metatheory of the core language—type safety for the entire language and consistency for the normalizing fragment—and provides a solid foundation. However, it is not feasible to write programs directly in the core language because the terms get cluttered with type annotations and type conversion proofs. This paper addresses the other half of the design: crafting a programmer-friendly *surface language*, which elaborates into the core.

The reason that elaboration is important in this context is that core ZOMBIE has a weak definition of equivalence. Most dependently-typed languages define terms to be equal when they are (at least) β-convertible. However, the presence of nontermination makes this definition awkward. To check whether two types are β-equivalent the type checker has to evaluate expressions inside them, which becomes problematic if expressions may diverge—what if the type checker gets stuck in an infinite loop? Existing languages fix an arbitrary global cut off for how many steps of evaluation the type-checker is willing to do (Cayenne [3]), or only reduce expressions that have passed a conservative termination test (Idris [9]). Core ZOMBIE, somewhat radically, omits automatic β-conversion completely. Instead, β-equality is available only through explicit conversion.

Because ZOMBIE does not include automatic β-conversion, it provides an opportunity to explore an alternative definition of equivalence in a surface language design.

Congruence closure, also known as the theory of equality with uninterpreted function symbols, is a basic operation in automatic theorem provers for first-order logic (particularly SMT solvers, such as Z3 [14]). Given some context Γ which contains assumptions of the form $a = b$, the congruence closure of Γ is the set of equations which are deducible by reflexivity, symmetry, transitivity, and changing subterms. Figure 1 specifies the congruence closure of a given context.

Although efficient algorithms for congruence closure are well-known [16, 22, 27] this reasoning principle has seen little use in *dependently-typed programming languages*. The problem is not lack of opportunity. Dependently-typed languages feature *propositional equality*, written $a = b$, which is a type that asserts the equality of the two expressions. Programs that use propositional equality build members of this type (using assumptions in the context, and various lemmas) and specify where and how they should be used. Congruence closure can assist with both of these tasks by automating the construction of these proofs and determining the "motive" for their elimination.

However, the adaption of this first-order technique to the higher-order logics of dependently-typed languages is not straightforward. The combination of congruence closure and full β-reduction makes the equality relation undecidable. As a result, most dependently-typed languages take the conservative approach of only incorporating congruence closure as a meta-operation, such as Coq's

$$\frac{a = b \in \Gamma}{\Gamma \vdash a = b} \qquad \frac{}{\Gamma \vdash a = a} \qquad \frac{\Gamma \vdash b = a}{\Gamma \vdash a = b}$$

$$\frac{\Gamma \vdash a = c \quad \Gamma \vdash c = b}{\Gamma \vdash a = b} \qquad \frac{\Gamma \vdash a = a' \quad \Gamma \vdash b = b'}{\Gamma \vdash a\ b = a'\ b'}$$

Figure 1. The "classic" congruence closure relation for untyped first-order logic terms

`congruence` tactic. While this tactic can assist with the creation of equality proofs, such proofs must still be explicitly eliminated. Proposals to use equations from the context automatically [1, 29, 30] have done so *in addition* to β-reduction, which makes it hard to characterize exactly which programs will typecheck, and also leaves open the question of how expressive congruence closure is in isolation.

In this work we define the ZOMBIE surface language to be fully "up to congruence", i.e. types which are equated by congruence closure can always be used interchangeably, and then show how the elaborator can implement this type system.

Designing a language around an elaborator—an unavoidably complicated piece of software—raises the risk of making the language hard to understand. Programmers could find it difficult to predict what core term a given surface term will elaborate to, or they may have to think about the details of the elaboration algorithm in order to understand whether a program will successfully elaborate at all.

We avoid these problems using two strategies. First, the syntax of the surface and the core language differ only by *erasable annotations* and the operational semantics ignores these annotations. Therefore the semantics of an expression is apparent just from looking at the source; the elaborator only adds annotations that can not change its behavior. Second, we define a *declarative specification* of the surface language, and prove that the elaborator is complete for the specification. As a result, the programmer does not have to think about the concrete elaboration algorithm.

We make the following contributions:

- We demonstrate how congruence closure is useful when programming, by comparing examples written in Agda, ZOMBIE, and ZOMBIE's explicitly-typed core language (Section 2).
- We define a dependently typed core language where the syntax contains erasable annotations (Section 3).
- We define a typed version of the congruence closure relation that is compatible with our core language, including features (erasure, injectivity, and generalized assumption) suitable for a dependent type system (Section 4).
- We specify the surface language using a bidirectional type system that uses this congruence closure relation as its *definition* of type equality (Section 5).
- We define an elaboration algorithm of the surface language to the core language (Section 6) based on a novel algorithm for typed congruence closure (Section 7). We prove that our elaboration algorithm is complete for the surface language and produces well-typed core language expressions. Our typed congruence closure algorithm both decides whether two terms are in the relation and also produces core language equality proofs.
- We have implemented these algorithms in ZOMBIE, extending the ideas of this paper to a language that includes datatypes and pattern matching, a richer logical fragment, and other features. Congruence closure works well in this setting; in particular, it significantly simplifies the typing rules for case-expressions (Section 8). Our implementation is available.[1]

```
data Term : Set where
  leaf : Term
  branch : Term → Term → Term
  var  : ℕ → Term

data Unify : (t1 t2 : Term) → Set where
  nomatch  : ∀{t1 t2} → Unify t1 t2
  match    : ∀{t1 t2} (s : Substitution)
        → ap s t1 ≡ ap s t2 → Unify t1 t2

-- Several lemmas --
apCompose : ∀ {s s'}  (t : Term)
          → ap (compose s s') t  ≡ ap s (ap s' t)
apCompose = ...

singleton-∉ : ∀ t x s →  (x ∈ t)
             → ap (singleton x s) t ≡ t
singleton-∉ = ...

varSingleton : ∀ x t → t ≡ ap (singleton x t) (var x)
varSingleton = ...
```

Figure 2. Supporting lemmas for Figure 3

2. Programming up to congruence

Consider this simple proof in Agda, which shows that zero is a right identity for addition.

```
npluszero : (n : Nat) → n + 0 ≡ n
npluszero zero    = refl
npluszero (suc m) = cong suc (npluszero m)
```

The proof follows by induction on natural numbers. In the base case, `refl` is a proof of `0 = 0`. In the next line, `cong` translates a proof of `m + 0 ≡ m` (from the recursive call) to a proof of `suc(m + 0) ≡ suc m`.

This proof relies on the fact that Agda's propositional equality relation (≡) is reflexive and a congruence relation. The former property holds by definition, but the latter must be explicitly shown. In other words, the proof relies on the following lemma:

```
cong : ∀ {A B} {m n : A}
       → (f : A → B) → m ≡ n → f m ≡ f n
cong f refl = refl
```

Now compare this proof to a similar result in ZOMBIE. The same reasoning is present: the proof follows via natural number induction, using the reduction behavior of addition in both cases.

```
npluszero : (n : Nat) → (n + 0 = n)
npluszero n =
  case n [eq] of
   Zero → (join : 0 + 0 = 0)
   Suc m →
     let _ = npluszero m in
       (join : (Suc m) + 0 = Suc (m + 0))
```

[1] `https://code.google.com/p/trellys/`

```
{-# NO_TERMINATION_CHECK #-}
unify : (t1 t2 : Term) → Unify t1 t2
unify leaf leaf = match empty refl
unify leaf (branch t2 t3) = nomatch
unify (branch t1 t2) leaf = nomatch
unify (branch t11 t12) (branch t21 t22)
      with unify t11 t21
...    | nomatch = nomatch
...    | match s p with unify (ap s t12) (ap s t22)
...               | nomatch = nomatch
...               | match s' q
  = match (compose s' s)
       (trans (apCompose (branch t11 t12))
         (trans (cong₂ (λ t1 t2 →
                    branch (ap s' t1) t2) p q)
            (sym (apCompose (branch t21 t22)))))
unify t1 (var x) with (x is∈ t1)
...              | no q
  = match (singleton x t1)
       (trans (singleton-∉ t x t q)
          (varSingleton x t))
...              | yes p with t
...                      | var y
  = match empty (cong var (sym (invvar p)))
...                      | _
  = nomatch
unify (var x) t2 with unify t2 (var x)
...              | nomatch   = nomatch
...              | match s p = match s (sym p)
```

```
prog unify : (t1 t2 : Term) → Unify t1 t2
rec unify t1 = \ t2 . case t1, t2 of
    leaf, leaf → match empty _
    leaf, branch _ _ → nomatch
    branch _ _, leaf → nomatch
    branch t11 t12, branch t21 t22 →
      case (unify t11 t21) of
        nomatch → nomatch
        match s p → case (unify (ap s t12) (ap s t22)) of
            nomatch → nomatch
            match s' _ →
              unfold (ap s' (ap s t1)) in
              unfold (ap s' (ap s t2)) in
              let _ = apCompose s' s t1 in
              let _ = apCompose s' s t2 in
                 match (compose s' s) _
    _ , var x → case (isin x t1) of
        no q  →
           let _ = varSingleton x t1 in
           let _ = singletonNotIn t1 x t1 q in
             match (singleton x t1) _
        yes _ → case t1 of
           var y → let [_] = invvar x y p in
                    match empty _
           _     →
                      nomatch
    var x, _ →  case (unify t2 (var x)) of
        nomatch → nomatch
        match s p → match s _
```

Figure 3. First-order unification in Agda (left) and in ZOMBIE (right)

Because ZOMBIE does not provide automatic β-equivalence, reduction must be made explicit above. The term `join` explicitly introduces an equality based on reduction. However, in the successor case, the ZOMBIE type checker is able to infer exactly how the equalities should be put together.

For comparison, the corresponding ZOMBIE core language term includes a number of explicit type coercions:

```
npluszero : (n : Nat) → (n + 0 = n)
npluszero (n : Nat) =
  case n [eq] of
   Zero → join [⇝ 0 + 0 = 0]
              ▷ join [~eq + 0 = ~eq]
   Suc m →
     let ih = npluszero m in
      join [⇝ (Suc m) + 0 = Suc (m + 0)]
         ▷ join [(Suc m) + 0 = Suc ~ih]
         ▷ join [~eq + 0 = ~eq]
```

Above, an expression of the form $a \triangleright b$ converts the type of the expression a, using the equality proof b. Equality proofs may be formed in two ways, either via co-reduction (if a_1 and a_2 both reduce to some common term b, then `join`[⇝ a_1 = a_2] is a proof of their equality) or by congruence (if a is a proof of b_1=b_2, then `join`[$\{\sim a/x\}A$] is a proof of $\{b_1/x\}A = \{b_2/x\}A$).

Both sorts of equality proofs are constructed in the example. In the base case, The proof `join[⇝ 0 + 0 = 0]` follows from reduction, and is converted to be a proof of `(n + 0) = n` by the congruence proof. Here, `eq` is a proof that `0 = n`, an assumption derived from pattern matching. Congruence reasoning constructs a proof that that `((0 + 0) = 0) = ((n + 0) = n)`; the parts that differ on each side of the equality are marked by `~eq` in the congruence proof. The successor case uses congruence twice. The equality derived from reduction is first coerced by a congruence derived from the recursive call (`ih : m + 0 = m`), so that it has type `((Suc m) + 0 = Suc m)`. This equality is then coerced by a congruence derived from `eq : (Suc m = n)`, so that the result has type `(n + 0) = n`.

As another example, Mu, Ko and Jansson [21] model relational program derivation in Agda. One property that they show is the universal property of the `foldr` function. In their code, they deliberately use Agda's features for equational reasoning—showing exactly the derivation of the equality. The reduction behavior of a program is an important part of a proof.

```
foldr-universal : {A B} (h : List A → B) f e →
    (h [] ≡ e) → (∀ x xs → h (x::xs) ≡ f x (h xs)) →
    (ys : List A) → h ys ≡ foldr f e ys
foldr-universal h f e base step [] = base
foldr-universal h f e base step (x :: xs) =
  h (x :: xs)
   ≡⟨step x xs⟩
  f x (h xs)
   ≡⟨cong (f x) (foldr-universal h f e base step xs)⟩
  f x (foldr f e xs)
   ≡⟨refl⟩
  foldr f e (x::xs)
 □
```

```
log snoc_inv : (xs ys : List A) → (z : A) → ((snoc xs z) = (snoc ys z)) → xs = ys
ind snoc_inv xs = \ ys z pf.  case xs [xeq], ys of
      Cons x xs' , Cons y ys' →
         let _ = (smartjoin : (snoc xs z) = Cons x (snoc xs' z)) in
         let _ = (smartjoin : (snoc ys z) = Cons y (snoc ys' z)) in
         let _ = snoc_inv xs' [ord xeq] ys' z _ in
           _
      ...

-- Agda pattern matching based solution
snoc-inv : ∀ xs ys z → (snoc xs z ≡ snoc ys z) → xs ≡ ys
snoc-inv (x :: xs')   (y :: ys') z pf with (snoc xs' z)  | (snoc ys' z)  | inspect (snoc xs') z | inspect (snoc ys') z
snoc-inv (.y :: xs')  (y :: ys') z refl    | .s2   | s2   | [ p ] | [ q ] with (snoc-inv xs' ys' z (trans p (sym q)))
snoc-inv (.y :: .ys') (y :: ys') z refl    | .s2   | s2   | [ p ] | [ q ]  | refl = refl
...

-- Alternative Agda solution based on congruence and injectivity
cons-inj1 : ∀ {x xs y ys} → ((x :: xs) ≡ y :: ys) → x ≡ y
cons-inj1 refl = refl

cons-inj2 : ∀ {x xs y ys} → x :: xs ≡ y :: ys → xs ≡ ys
cons-inj2 refl = refl

snoc-inv' : ∀ xs ys z → (snoc xs z ≡ snoc ys z) → xs ≡ ys
snoc-inv' (x :: xs') (y :: ys') z pf = cong₂ _::_ (cons-inj1 pf) (snoc-inv' xs' ys' z (cons-inj2 pf))
...
```

Figure 4. Pattern matching can be tricky in Agda

In ZOMBIE, the congruence closure algorithm can put the various steps together, including the unfolding, the step case and the induction. Note that ZOMBIE allows programmers to notate arguments that are irrelevant (in square brackets, should have no affect on computation) and inferred (⇒ instead of →, automatically determined through unification).

```
foldrUniversal : [A :Type]⇒ [B:Type] ⇒
  (h : List A → B) → (f : A → B → B) →
  (e : B) → (h [] = e) →
  ((x:A) → (xs:List A) → h (x :: xs) = f x (h xs)) →
  (ys : List A) → h ys = fold e f ys
foldrUniversal [A][B] h f e base step xs =
 case xs [_] of
  [] → let _ = (join : fold e f [] = e) in
        _
  x :: xs' →
    let _ = step x xs' in
    let _ = foldrUniversal h f e base step xs' in
    let _ = (join : f x (fold e f xs')
                  = fold e f (x :: xs')) in
    _
```

For a larger example, consider unification of first-order terms (Figure 3). For this example, the term language is the simplest possible, consisting only of binary trees constructed by `branch` and `leaf` and possibly containing unification variables, `var`, represented as natural numbers. We also use a type `Substitution` of substitutions, which are built by the functions `singleton` and `compose`, and applied to terms by `ap`.

Proving that `unify` terminates is difficult because the termination metric involves not just the structure of the terms but also the number of unassigned unification variables. (For example, see McBride [20]). To save development effort, a programmer may elect to prove only a partial correctness property: *if* the function terminates then the substitution it returns is a unifier.

In other words, if the `unify` function returns, it either says that the terms do not match, or produces a substitution `s` and a proof that `s` unifies them. We write the data structure in ZOMBIE as follows (the Agda version is similar):

```
data Unify (t1 : Term) (t2 : Term) : Type where
  nomatch
  match of (s : Substitution) (pf : ap s t1 = ap s t2)
```

Comparing the Agda and ZOMBIE implementations, we can see the effect of programming up-to-congruence instead of up-to-β. When the unifier returns `match`, it needs to supply a proof of equality. The Agda version explicitly constructs the proof using equational reasoning, which involves calling congruence lemmas `sym`, `trans`, `cong` and $\texttt{cong}_2$ from the standard library. The ZOMBIE version leaves such proof arguments as just an underscore, meaning that it can be inferred from the equations in the context. For that purpose, it introduces equalities to the context with `unfold` (for β-reductions, see Section 8.2) and with calls to relevant lemmas.

Figure 4 demonstrates how congruence closure makes ZOMBIE's version of dependently-typed pattern matching (i.e *smart case*) both simple and powerful. The figure compares (parts of) inductive proofs in ZOMBIE and Agda of an inversion lemma about the `snoc` operation, which appends an element to the end of a list. When both lists are nonempty, the proof argument can be used to derive that `x = y` (using the injectivity of `Cons`), and the recursive call shows that `xs' = ys'`. Congruence closure both puts these together in a proof of `Cons x xs' = Cons y ys'` and supplies the necessary proof for the recursive call.

In Agda, one is tempted to prove the property by pattern matching on the equality between the lists. This approach leads to a

$$
\begin{array}{rcl}
x, y, f, g, h & \in & \text{expression variables} \\
\\
\text{expressions} & & \\
a, b, c, A, B & ::= & \mathsf{Type} \mid x \\
& \mid & (x\!:\!A) \to B \mid \mathsf{rec}\ f_A\ x.a \mid a\ b \\
& \mid & \bullet(x\!:\!A) \to B \mid \mathsf{rec}\ f_A\ \bullet_x.a \mid a\ \bullet_b \\
& \mid & a = b \mid \mathsf{join}_\Sigma \mid a_{\triangleright b} \\
\\
\text{strategies} & & \\
\Sigma & ::= & \leadsto_{\mathsf{p}}\ i\ j : a = b \\
& \mid & \{\sim v_1/x_1\} \ldots \{\sim v_j/x_j\} c : B \\
& \mid & \mathsf{injdom}\ a \mid \mathsf{injrng}\ a\ b \mid \mathsf{injeq}\ i\ a \\
\\
\text{values} & & \\
v & ::= & \mathsf{Type} \mid x \\
& \mid & (x\!:\!A) \to B \mid \mathsf{rec}\ f_A\ x.a \\
& \mid & \bullet(x\!:\!A) \to B \mid \mathsf{rec}\ f_A\ \bullet_x.a \\
& \mid & a = b \mid \mathsf{join}_\Sigma \mid v_{\triangleright b}
\end{array}
$$

Figure 5. Syntax

$\boxed{a \leadsto_{\mathsf{cbv}} b}$

$$\frac{}{(\mathsf{rec}\ f\ x.a)\ v \leadsto_{\mathsf{cbv}} \{v/x\}\,\{\mathsf{rec}\ f\ x.a/f\}\ a}\ \textsc{SCAppBeta}$$

$$\frac{}{(\mathsf{rec}\ f\ \bullet.a)\ \bullet \leadsto_{\mathsf{cbv}} \{\mathsf{rec}\ f\ \bullet.a/f\}\ a}\ \textsc{SCIAppBeta}$$

$$\frac{a \leadsto_{\mathsf{cbv}} a'}{a\ b \leadsto_{\mathsf{cbv}} a'\ b}\ \textsc{SCctx1}$$

$$\frac{a \leadsto_{\mathsf{cbv}} a'}{v\ a \leadsto_{\mathsf{cbv}} v\ a'}\ \textsc{SCctx2}$$

$$\frac{a \leadsto_{\mathsf{cbv}} a'}{a\ \bullet \leadsto_{\mathsf{cbv}} a'\ \bullet}\ \textsc{SCctx3}$$

Figure 6. Call-by-value operational semantics

"quite fun" puzzle.[2] Here, the equivalence between `x` and `y` cannot be observed until `(snoc xs' z)` and `(snoc xs' z)` are named. The so-called "inspect on steroids" trick provides the equalities `(p : (snoc xs' z = s2)` and `q : (snoc ys' z) = s2)` that are necessary to constructing the fourth argument for the recursive call. Although this development is not long, it is not at all straightforward, requiring advanced knowledge of Agda idioms.

Alternatively, the reasoning used in the ZOMBIE example is also available in Agda, as in the definition of `snoc-inv'`. However, this version requires the use of helper functions to prove that cons is injective and congruent.

3. Annotated core language

We now turn to the theory of the system. We begin by describing the target of the elaborator: our annotated core language. This language is a small variant of the dependently-typed call-by-value language defined in prior work [28]. It corresponds to a portion of ZOMBIE's core language, but to keep the proofs tractable we omit ZOMBIE's recursive datatypes and replace its terminating sublanguage [10] with syntactic value restrictions.

[2] Posed by Eric Mertens on `#agda`.

$$
\begin{array}{lcl}
|\mathsf{Type}| & = & \mathsf{Type} \\
|x| & = & x \\
|\mathsf{rec}\ f_A\ x.a| & = & \mathsf{rec}\ f\ x.|a| \\
|\mathsf{rec}\ f_A\ \bullet_x.a| & = & \mathsf{rec}\ f\ \bullet.|a| \\
|(x\!:\!A) \to B| & = & (x\!:\!|A|) \to |B| \\
|a\ b| & = & |a|\ |b| \\
|\bullet\,(x\!:\!A) \to B| & = & \bullet(x\!:\!|A|) \to |B| \\
|a\ \bullet_b| & = & |a|\ \bullet \\
|a = b| & = & (|a| = |b|) \\
|\mathsf{join}_\Sigma| & = & \mathsf{join} \\
|a_{\triangleright b}| & = & |a|
\end{array}
$$

Figure 7. The erasure function $|\cdot|$

The syntax is shown in Figure 5. Terms, types and the sort Type are collapsed into one syntactic category. By convention, we use lowercase metavariables a, b for expressions that are terms and uppercase metavariables A, B for expressions that are types.We use the notation $\{a/x\}\ B$ to denote the capture-avoiding substitution of a for x in B. The notation $\mathsf{FV}\ (a)$ calculates the set of free variables appearing in an expression. As is standard in dependently-typed languages, our notation for nondependent function types $A \to B$ is syntactic sugar for function types $(x\!:\!A) \to B$, where the variable x is not free in B.

Type *annotations*, such as A in $\mathsf{rec}\ f_A\ x.a$, are optional and may be omitted from expressions. Annotations are subscripted in Figure 5. The meta-operator $|a|$ removes these annotations (Figure 7). Expressions that contain no typing annotations are called *erased*.

An expression that includes all annotations is called a *core* or *annotated* expression. The core typing judgement, written $\Gamma \vdash a : A$ and described below, requires that all annotations be present. In this case, the judgement is syntax-directed and trivially decidable. Given a context and an expression, there is a simple algorithm to determine the type of that expression (if any). In contrast, type checking for erased terms is undecidable.

The only role of annotations is to ensure decidable type checking. They have no effect on the semantics. In fact, the operational semantics, written $a \leadsto_{\mathsf{cbv}} b$, is defined only for erased terms and extended to terms with annotations via erasure. This operational semantics is a small-step, call-by-value evaluation relation, shown in Figure 6.

Figure 8 shows the typing rules of the core language typing judgement $\Gamma \vdash a : A$. The type system identifies which expressions are types: A is a type when $\Gamma \vdash A : \mathsf{Type}$ holds. The rule TTYPE ensures that Type itself is typeable. Additionally, the judgement $\vdash \Gamma$ (elided from the figure) states that each type in Γ is well-formed.

The variable lookup rule TVAR is standard. Function types are formed by the rule TPI, and introduced and eliminated by TREC, TAPP, and TDAPP. We use the notation $A \to B$ for $(x\!:\!A) \to B$ when x is not free in B.

Recursive functions are defined using expressions $\mathsf{rec}\ f\ x.a$, with the typing rule TREC. Such expressions are values, and applications step by the rule $(\mathsf{rec}\ f\ x.a)\ v \leadsto_{\mathsf{cbv}} \{v/x\}\,\{\mathsf{rec}\ f\ x.a/f\}\ a$. If the function makes no recursive calls we also use the syntactic sugar $\lambda x.a$. When a function has a dependent type (TDAPP) then its argument must be a value (this restriction is common for languages with nontermination [18, 32]).

Irrelevance In addition to the normal function type, the core language also include *computationally irrelevant* function types $\bullet(x :$

$\boxed{\Gamma \vdash a : A}$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}}\,\textsc{TType} \qquad \frac{x : A \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash x : A}\,\textsc{TVar} \qquad \frac{\begin{array}{l}\Gamma \vdash A : \mathsf{Type}\\ \Gamma, x : A \vdash B : \mathsf{Type}\end{array}}{\Gamma \vdash (x\!:\!A) \to\ B : \mathsf{Type}}\,\textsc{TPi}$$

$$\frac{\begin{array}{l}\Gamma \vdash (x\!:\!A_1) \to\ A_2 : \mathsf{Type}\\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash a : A_2\end{array}}{\Gamma \vdash \mathsf{rec}\ f_{(x A_1) \to\ A_2}\ x.a : (x\!:\!A_1) \to\ A_2}\,\textsc{TRec} \qquad \frac{\begin{array}{l}\Gamma \vdash a : A \to B\\ \Gamma \vdash b : A\end{array}}{\Gamma \vdash a\ b : B}\,\textsc{TApp} \qquad \frac{\begin{array}{l}\Gamma \vdash a : (x\!:\!A) \to\ B\\ \Gamma \vdash v : A\end{array}}{\Gamma \vdash a\ v : \{v/x\}\, B}\,\textsc{TDApp}$$

$$\frac{\begin{array}{l}\Gamma \vdash A : \mathsf{Type}\\ \Gamma, x : A \vdash B : \mathsf{Type}\end{array}}{\Gamma \vdash \bullet(x\!:\!A) \to\ B : \mathsf{Type}}\,\textsc{TIPi} \qquad \frac{\begin{array}{l}\Gamma \vdash \bullet(x\!:\!A_1) \to\ A_2 : \mathsf{Type}\\ \Gamma, f : \bullet(x\!:\!A_1) \to\ A_2, x : A_1 \vdash a : A_2\\ x \notin \mathsf{FV}\,(|a|)\end{array}}{\Gamma \vdash \mathsf{rec}\ f_{(x A_1) \to\ A_2}\ \bullet_x.a : (x\!:\!A_1) \to\ A_2}\,\textsc{TIRec} \qquad \frac{\begin{array}{l}\Gamma \vdash a : \bullet(x\!:\!A) \to\ B\\ \Gamma \vdash v : A\end{array}}{\Gamma \vdash a\ \bullet_v : \{v/x\}\, B}\,\textsc{TIDApp}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash a = b : \mathsf{Type}}\,\textsc{TEq} \qquad \frac{\Gamma \vdash a : A \quad \Gamma \vdash v : A = B \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash a_{\triangleright v} : B}\,\textsc{TCast}$$

$$\frac{|a_1| \leadsto^i_{\mathsf{p}} b \quad |a_2| \leadsto^j_{\mathsf{p}} b \quad \Gamma \vdash a_1 = a_2 : \mathsf{Type}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{p}}\, i\, j : a_1 = a_2} : a_1 = a_2}\,\textsc{TJoinP} \qquad \frac{|a_1| \leadsto^i_{\mathsf{cbv}} b \quad |a_2| \leadsto^j_{\mathsf{cbv}} b \quad \Gamma \vdash a_1 = a_2 : \mathsf{Type}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}}\, i\, j : a_1 = a_2} : a_1 = a_2}\,\textsc{TJoinC}$$

$$\frac{\begin{array}{l}\Gamma \vdash B : \mathsf{Type} \quad \forall k.\ \Gamma \vdash v_k : a_k = b_k\\ |B| = |(\{a_1/x_1\} \ldots \{a_j/x_j\}\, c = \{b_1/x_1\} \ldots \{b_j/x_j\}\, c)|\end{array}}{\Gamma \vdash \mathsf{join}_{\{\sim v_1/x_1\} \ldots \{\sim v_j/x_j\}\, c : B} : B}\,\textsc{TJSubst} \qquad \frac{\Gamma \vdash v : (A_1 = A_2) = (B_1 = B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injeq}\ i\ v} : A_i = B_i}\,\textsc{TJInjEq}$$

$$\frac{\Gamma \vdash v : ((x\!:\!A_1) \to\ B_1) = ((x\!:\!A_2) \to\ B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injdom}\ v} : A_1 = A_2}\,\textsc{TJInjDom} \qquad \frac{\Gamma \vdash v_1 : ((x\!:\!A) \to\ B_1) = ((x\!:\!A) \to\ B_2) \quad \Gamma \vdash v_2 : A}{\Gamma \vdash \mathsf{join}_{\mathsf{injrng}\ v_1\ v_2} : \{v_2/x\}\, B_1 = \{v_2/x\}\, B_2}\,\textsc{TJInjRng}$$

$$\frac{\Gamma \vdash v : (\bullet(x\!:\!A_1) \to\ B_1) = (\bullet(x\!:\!A_2) \to\ B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injdom}\ v} : A_1 = A_2}\,\textsc{TJIInjDom} \qquad \frac{\Gamma \vdash v_1 : (\bullet(x\!:\!A) \to\ B_1) = (\bullet(x\!:\!A) \to\ B_2) \quad \Gamma \vdash v_2 : A}{\Gamma \vdash \mathsf{join}_{\mathsf{injrng}\ v_1\ v_2} : \{v_2/x\}\, B_1 = \{v_2/x\}\, B_2}\,\textsc{TJIInjRng}$$

Figure 8. Typing rules for the annotated core language

$A) \to\ B$, which are inhabited by irrelevant functions rec $f_A\ \bullet_x.b$ and eliminated by irrelevant applications $a\ \bullet_b$. Many expressions in a dependently typed program are only used for type checking, but do not affect the runtime behavior of the program, and these can be marked irrelevant. Some common examples are type arguments to polymorphic functions (e.g. $\mathsf{map} : \bullet(A\ B : \mathsf{Type}) \to (A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B)$, indices to dependent datatypes (e.g. $\mathsf{append} : \bullet(n\,m : \mathsf{Nat}) \to \mathsf{Vec}\, n \to \mathsf{Vec}\, m \to \mathsf{Vec}\ (n{+}m)$), and preconditions to functions (e.g. $\mathsf{divide} : (x\,y : \mathsf{Int}) \to \bullet(y \neq 0) \to \mathsf{Int}$). When comparing two expressions for equality the typechecker will ignore arguments in irrelevant positions. For example, if both p and q have type $y \neq 0$, the typechecker will treat $\mathsf{divide}\ x\ y\ \bullet_p$ and $\mathsf{divide}\ x\ y\ \bullet_q$ as equal without looking at the last argument.

Our treatment of irrelevance follows ICC* [6]. The introduction rule for irrelevant functions, TIREC, is similar to the rule for normal functions, but with the additional restriction that the bound variable must not remain in the erasure of the body b. This restriction means that x can only appear in irrelevant positions in b, such as type annotations and proofs for conversions.

The application rule TIDAPP is also similar to normal application. To ensure type safety we must ensure that diverging terms are not erased [28]. In this version of the core language we use a rather simple-minded restriction, by requiring the argument to be a value. A more ambitious alternative would be to make the type system distinguish between terminating and possibly nonterminating expressions, like in [10].

We include computational irrelevance in this work to show that, besides being generally useful, irrelevance works well with congruence closure. Given that we already handle erasable annotations, we can support full irrelevance for free.

Equality The typing rules at the bottom of Figure 8 deal with propositional equality, a primitive type. The formation rule TEQ states $a = b$ is a well-formed type whenever a and b are two well-typed expressions. There is no requirement that they have the *same* type (that is to say, our equality type is heterogeneous).

Propositional equality is eliminated by the rule TCAST: given a proof, v of an equation $A = B$ we can change the type of an expression from A to B. Since our equality is heterogeneous, we need to check that B is in fact a type. We require the proof to be a value in order to rule out divergence. A full-scale language could use a more ambitious termination analysis. (Indeed, our ZOMBIE implementation does so.) However, the congruence proofs generated by our elaborator are syntactic values, so for the purposes of this paper, the simple value restriction is enough. The proof term v in a type cast is an erasable annotation with no operational significance, so the typechecker considers equalities like $a = a_{\triangleright v}$ to be trivially true, and the elaborator is free to insert coercions using congruence closure proofs anywhere.

The rest of the figure shows introduction rules for equality. Equality proofs do not carry any information at runtime, so they all use the same term constructor join, but with different (erasable) annotations, Σ.

The rule TJOINP introduces equations which are justified by the operational semantics. ZOMBIE source programs must use TJOINP to explicitly indicate expressions that should be reduced. The rule states that join is a proof of $a_1 = a_2$ when the erasures of a_1 and a_2 reduce to a common expression b, using the parallel reduction relation. This common expression, b, is not required to be a value. Note that without normalization, we need a cutoff for how long to evaluate, so programmers must specify the number of steps i, j of reduction to allow (in ZOMBIE this defaults to 1000 if these numbers are elided). The rule TJOINC is similar, except that it uses call-by-value evaluation directly instead of parallel reduction.

Actually, these two rules hint at some subtleties which are outside the scope of this paper. In a normalizing confluent language, the evaluation order does not matter. But in a language with nontermination the programmer needs more fine-grained control. Our implementation currently offers two evaluation strategies: CBV evaluation (good for cases where an expression is expected to reduce to a value), and a parallel reduction which heuristically avoids unfolding recursive calls inside a function body (good when trying to prove recursive fixpoint equations).

The rule TJSUBST states that equality is a congruence. The simplest use of the rule is to change a single subexpression, using a proof v. The use of the proof is marked with a tilde in the Σ annotation; for example, if $\Gamma \vdash v : y = 0$ then we can prove the equality $\mathsf{join}_{\mathsf{Vec\ Nat}\ (\sim v) : \mathsf{Vec\ Nat}\ y = \mathsf{Vec\ Nat}\ 0}$. One can also eliminate several different equality proofs in one use of the rule. For example if $\Gamma \vdash v_1 : x = 0$ and $\Gamma \vdash v_2 : y = 0$, then we can use both proofs at once in the expression $\mathsf{join}_{\mathsf{Vec\ Nat}\ (\sim v_1 + \sim v_2)} : \mathsf{Vec\ Nat}\ (x + y) = \mathsf{Vec\ Nat}\ (0+0)$. The syntax of subst includes a type annotation B, and the last premise of the TJSUBST rule checks that the ascribed type B matches what one gets after substituting the given equalities into the template c. This annotation adds flexibility because the check is only up-to erasure: if needed the programmer can give the left- and right-hand side of B different annotations to make both sides well-typed.

Finally, the rules TJINJEQ, TJINJDOM, TJINJDOM, TJINJRNG, and TJIINJRNG state that the equality type and arrow type constructors are injective. The rule for arrow domains is exactly what one would expect: if $(x:A) \to B = (x:A') \to B$, then $A = A'$. The rule for arrow codomains must account for the bound variable x, so it states that the codomains are equal when any value v is substituted in. Making type constructors injective is unconventional for a dependent language. It is incompatible with e.g. Homotopy Type Theory, which proves $\mathsf{Nat} \to \mathsf{Void} = \mathsf{Bool} \to \mathsf{Void}$. However, in our language we need arrow injectivity to prove type preservation, because type casts are erased and do not block reduction [28]. For example, if a function coerced by type cast steps via β-reduction, we must use arrow injectivity to derive casts for the argument and result of the application.

We also add injectivity for the equality type constructor (TJINJEQ). This is not required for type safety, but it *is* justified by the metatheory, so it is safe to add. Injectivity is important for the surface language design, see Section 6.

The core language satisfies the usual properties for type systems. For the proofs in Section 6 we rely on the fact that it satisfies weakening, substitution (restricted to values), and regularity.

***Lemma* 1** (Weakening)**.** If $\Gamma \vdash a : A$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash a : A$.

***Lemma* 2** (Value Substitution)**.** If $\Gamma, x : A \vdash b : B$ and $\Gamma \vdash v : A$, then $\Gamma \vdash \{v/x\}\, b : \{v/x\}\, B$.

***Lemma* 3** (Regularity)**.**

1. $\vdash \Gamma$ and $x : A \in \Gamma$, then $\Gamma \vdash A : \mathsf{Type}$.
2. If $\Gamma \vdash a : A$ then $\vdash \Gamma$ and $\Gamma \vdash A : \mathsf{Type}$

It also satisfies preservation, progress, and decidable type checking. The proofs of these lemmas are in Sjöberg et al. [28].

4. Congruence closure

The driving idea behind our surface language is that the programmer should never have to explicitly write a type cast $a_{\triangleright v}$ if the proof v can be inferred by congruence closure. In this section we exactly specify which proofs can be inferred, by defining the typed congruence closure relation $\Gamma \vDash a = b$ shown in Figure 9.

Like the usual congruence closure relation for first-order terms, the rules in Figure 9, specify that this relation is reflexive, symmetric and transitive. It also includes rules for using assumptions in the context and congruence by changing subterms. However, we make a few changes:

First, we add typing premises (in TCCREFL and TCCERASURE) to make sure that the relation only equates well-typed and fully-annotated core language terms. In other words,

If $\Gamma \vDash a = b$, then $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$.

Next, we adapt the congruence rule so that it corresponds to the TJSUBST rule of the core language. In particular, the rule TCCCONGRUENCE includes an explicit erasure step so that the two sides of the equality can differ in their erasable portions.

Furthermore, we extend the relation in several ways.[3] We automatically use computational irrelevance, in the rule TCCERASURE. This makes sure that the programmer can ignore all annotations when reasoning about programs. Also, we reason up to injectivity of datatype constructors (in rules TCCINJDOM, TCCINJRNG, and TCCINJEQ). As mentioned in Section 3 these rules are valid in the core language, and we will see in Section 6 that there is good reason to make the congruence closure algorithm use them automatically. Note that we restrict rule TCCINJRNG so that it applies only to nondependent function types; we explain this restriction in Section 6.

Finally, the rule TCCASSUMPTION is a bit stronger than the classic rule from first order logic. In the first-order logic setting, this rule is defined as just the closure over equations in the context:

$$\frac{x : a = b \in \Gamma}{\Gamma \vDash a = b}$$

However, in a dependently typed language, we can have equations between equations. In this setting, the classic rule does not respect CC-equivalence of contexts. For example, it would prove the first of the following two problem instances, but not the second.

$$x : \mathsf{Nat}, y : \mathsf{Nat}, a : \mathsf{Type}, h_1 : (x = y) = a, h_2 : x = y \vDash x = y$$

$$x : \mathsf{Nat}, y : \mathsf{Nat}, a : \mathsf{Type}, h_1 : (x = y) = a, h_2 : a \qquad \vDash x = y$$

Therefore we replace the rule with the stronger version shown in the figure.

We were led to these strengthened rules by theoretical considerations when trying to show that our elaboration algorithm was complete with respect to the declarative specification (see Section 6). Once we implemented the current set of rules, we found that they were useful in practice as well as in theory, because they improved the elaboration of some examples in our test suite.

[3] Systems based around congruence closure often strengthen their automatic theorem prover in some way, e.g. Nieuwenhuis and Oliveras [23] add reasoning about natural number equations, and the Coq `congruence` tactic automatically uses injectivity of data constructors [12].

$$\frac{\Gamma \vdash a : A}{\Gamma \vDash a = a}\textsc{TCCRefl} \qquad \frac{\Gamma \vDash a = b}{\Gamma \vDash b = a}\textsc{TCCSym} \qquad \frac{\Gamma \vDash a = b \quad \Gamma \vDash b = c}{\Gamma \vDash a = c}\textsc{TCCTrans}$$

$$\frac{|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vDash a = b}\textsc{TCCErasure} \qquad \frac{x : A \in \Gamma \quad \Gamma \vDash A = (a = b)}{\Gamma \vDash a = b}\textsc{TCCAssumption}$$

$$\frac{\begin{array}{l}\Gamma \vdash A = B : \mathsf{Type} \quad \forall k.\ \Gamma \vDash a_k = b_k \\ |A = B| = |\{a_1/x_1\} \ldots \{a_j/x_j\}\, c = \{b_1/x_1\} \ldots \{b_j/x_j\}\, c|\end{array}}{\Gamma \vDash A = B}\textsc{TCCCongruence} \qquad \frac{\Gamma \vDash (a_1 = a_2) = (b_1 = b_2)}{\Gamma \vDash a_k = b_k}\textsc{TCCInjEq}$$

$$\frac{\Gamma \vDash ((x{:}A_1) \to B_1) = ((x{:}A_2) \to B_2)}{\Gamma \vDash A_1 = A_2}\textsc{TCCInjDom} \qquad \frac{\Gamma \vDash (A_1 \to B_1) = (A_2 \to B_2)}{\Gamma \vDash B_1 = B_2}\textsc{TCCInjRng}$$

$$\frac{\Gamma \vDash (\bullet(x{:}A_1) \to B_1) = (\bullet(x{:}A_2) \to B_2)}{\Gamma \vDash A_1 = A_2}\textsc{TCCIInjDom} \qquad \frac{\Gamma \vDash (\bullet A_1 \to B_1) = (\bullet A_2 \to B_2)}{\Gamma \vDash B_1 = B_2}\textsc{TCCIInjRng}$$

Figure 9. Typed congruence closure relation

The stronger assumption rule is useful in situations where type-level computation produces equality types, for example when using custom induction principles. Say we want to prove a theorem $\forall n.f(n) = g(n)$ by first proving that course-of-values induction holds for any predicate $P : \mathsf{Nat} \to \mathsf{Type}$, and then instantiating the induction lemma with $P := (\lambda n.f(n) = g(n))$. Then in the step case after calling the induction hypothesis on some number m, the context will contain $H : P(m)$, and by β-reduction we know that $P(m) = (f(m) = g(m))$. In that situation, the extended assumption rule says that H should be used when constructing the congruence closure of the context, even if the programmer does not apply an explicit type cast to H, which accords with intuition.

5. Surface language

Next, we give a precise specification of the surface language, which shows how type inference can use congruence closure to infer casts of the form $a_{\triangleright v}$. Note that this process involves determining both the location of such casts and the proof of equality v.

Figure 10 defines a *bidirectional type system* for a partially annotated language. This type system is defined by two (mutually defined) judgements: *type synthesis*, written $\Gamma \vdash a \Rightarrow A$, and *type checking*, written $\Gamma \vdash a \Leftarrow A$. Here Γ and a are always inputs, but A is an output of the synthesizing judgement and an input of the checking judgement.

Bidirectional systems are a standard form of local type inference. In such systems, the programmer must provide types for top-level definitions, and those definitions are then *checked* against the ascribed types. As a result, most type annotations can be omitted, e.g. in a definition like

```
foo : Nat → Nat → Nat
foo = \ x y . 2*x + y
```

there is no need for type annotations on the bound variables x and y, since the function is checked against a known top-level type.

Most rules of this type system are standard for bidirectional systems [25], including the rules for inferring the types of variables (IVAR), the well-formedness of types (IEQ, ITYPE, and IPI), non-dependent application (IAPP), and the mode switching rules CINF and IANNOT. Any term that has enough annotations to synthesize a type A also checks against that type (CINF). Conversely, some terms (e.g. functions) require a known type to check against, and so if the surrounding context does not specify one, the programmer must add a type annotation (IANNOT).

The rules ICAST and CCAST in Figure 10 specify that checking and inference work "up-to-congruence." At any point in the typing derivation, the system can replace the inferred or checked type with something congruent. The notation $\Gamma \vDash^{\exists} A = B$ lifts the congruence closure judgement from Section 4 to the partially annotated surface language. These two rules contain kinding premises to maintain well-formedness of types. The invariant maintained by the type system is that (in a well-formed context Γ) any synthesized type is guaranteed to be well-kinded, while it is the caller's responsibility to ensure that any time the checking judgement is used the input type is well-kinded.

The rule for checking functions (CREC) is almost identical to the corresponding rule in the core language, with just two changes. First, the programmer can omit the types A_1, and A_2, because in a bidirectional system they can be deduced from the type the expression is checked against. Second, the new premise injrng slightly restricts the use of this rule. The difficulty is that the congruence closure algorithm does not implement the full TJINJRNG rule of the core language, but injective reasoning is needed by the type checker. Therefore, we rule out function types that do not support injectivity for their ranges in certain (pathological) typing contexts. This premise also appears in the rule for dependent application (IDAPP). We return to this issue in Section 6.

Equations that are provable via congruence closure are available via the checking rule, CREFL. In this case the proof term is just join, written as an underscore in the concrete syntax. Because this is a checking rule, the equation to be proved does not have to be written down directly if it can be inferred from the context.

The rule IJOINP proves equations using the operational semantics. We saw this rule used in the `npluszero` example, written `join : 0 + 0 = 0` in the concrete syntax. Note that the programmer must explicitly write down the terms that should be reduced. The rule IJOINP is a synthesizing rather than checking rule in order to ensure that the typing rules are effectively implementable. Although the type system works "up to congruence" the operational semantics do not. So the expression itself needs to contain enough information to tell the typechecker which member of the equivalence class should be reduced—it cannot get this information from the checking context. (In practice, having to explicitly write this annotation can be annoying. The ZOMBIE implementation includes a feature `smartjoin` which can help—see Section 8.2).

$$\boxed{\Gamma \vdash a \Rightarrow A}$$

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \models^{\exists} A = B \quad \Gamma \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash a \Rightarrow B}\ \textsc{ICast}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma \vdash a \Leftarrow A}{\Gamma \vdash a_A \Rightarrow A}\ \textsc{IAnnot}$$

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \vdash b \Rightarrow B}{\Gamma \vdash a = b \Rightarrow \mathsf{Type}}\ \textsc{IEq}$$

$$\frac{\begin{array}{c}\Gamma \vdash a_1 = a_2 \Leftarrow \mathsf{Type}\\ |a_1| \leadsto_{\mathsf{p}}^{i} b \quad |a_2| \leadsto_{\mathsf{p}}^{j} b\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{p}}\, i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\ \textsc{IJoinP}$$

$$\frac{\vdash \Gamma \Leftarrow}{\Gamma \vdash \mathsf{Type} \Rightarrow \mathsf{Type}}\ \textsc{IType}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma, x : A \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash \bullet(x : A) \rightarrow B \Rightarrow \mathsf{Type}}\ \textsc{IIPi}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow \bullet(x : A) \rightarrow B\\ \Gamma \vdash v \Leftarrow A\\ \Gamma \models^{\exists} \mathsf{injrng}\ \bullet(x : A) \rightarrow B\ \mathsf{for}\ v\\ \Gamma \vdash \{v_A/x\}\, B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a \bullet_v \Rightarrow \{v_A/x\}\, B}\ \textsc{IIDApp}$$

$$\frac{\begin{array}{c}\Gamma \vdash a_1 = a_2 \Leftarrow \mathsf{Type}\\ |a_1| \leadsto_{\mathsf{cbv}}^{i} b \quad |a_2| \leadsto_{\mathsf{cbv}}^{j} b\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}}\, i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\ \textsc{IJoinC}$$

$$\frac{\vdash \Gamma \Leftarrow \quad x : A \in \Gamma \quad \Gamma \vdash A \Leftarrow \mathsf{Type}}{\Gamma \vdash x \Rightarrow A}\ \textsc{IVar}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma, x : A \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash (x : A) \rightarrow B \Rightarrow \mathsf{Type}}\ \textsc{IPi}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow A \rightarrow B \quad \Gamma \vdash b \Leftarrow A\\ \Gamma \vdash B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a\ b \Rightarrow B}\ \textsc{IApp}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow (x : A) \rightarrow B \quad \Gamma \vdash v \Leftarrow A\\ \Gamma \models^{\exists} \mathsf{injrng}\ (x : A) \rightarrow B\ \mathsf{for}\ v\\ \Gamma \vdash \{v_A/x\}\, B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a\ v \Rightarrow \{v_A/x\}\, B}\ \textsc{IDApp}$$

$$\boxed{\Gamma \vdash a \Leftarrow A}$$

$$\frac{\Gamma \vdash a \Leftarrow A \quad \Gamma \models^{\exists} A = B \quad \Gamma \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash a \Leftarrow B}\ \textsc{CCast}$$

$$\frac{\Gamma \vdash a \Rightarrow A}{\Gamma \vdash a \Leftarrow A}\ \textsc{CInf}$$

$$\frac{\Gamma \models^{\exists} a = b}{\Gamma \vdash \mathsf{join} \Leftarrow a = b}\ \textsc{CRefl}$$

$$\frac{\begin{array}{l}\Gamma, f : (x : A_1) \rightarrow A_2, x : A_1 \vdash a \Leftarrow A_2\\ \Gamma, f : (x : A_1) \rightarrow A_2, x : A_1 \models^{\exists} \mathsf{injrng}\ (x : A_1) \rightarrow A_2\ \mathsf{for}\ x\\ \Gamma, f : (x : A_1) \rightarrow A_2 \vdash (x : A_1) \rightarrow A_2 \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash \mathsf{rec}\ f\ x.a \Leftarrow (x : A_1) \rightarrow A_2}\ \textsc{CRec}$$

$$\frac{\begin{array}{l}\Gamma, f : \bullet(x : A_1) \rightarrow A_2, x : A_1 \vdash a \Leftarrow A_2\\ x \notin \mathsf{FV}\,(|a|)\\ \Gamma, f : \bullet(x : A_1) \rightarrow A_2, x : A_1 \models^{\exists} \mathsf{injrng}\ \bullet(x : A_1) \rightarrow A_2\ \mathsf{for}\ x\\ \Gamma, f : \bullet(x : A_1) \rightarrow A_2 \vdash \bullet(x : A_1) \rightarrow A_2 \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash \mathsf{rec}\ f\ \bullet.a \Leftarrow \bullet(x : A_1) \rightarrow A_2}\ \textsc{CIRec}$$

$$\boxed{\Gamma \models^{\exists} a = b}$$

$$\frac{\Gamma' \models a' = b' \quad |a| = |a'| \quad |b| = |b'| \quad |\Gamma| = |\Gamma'|}{\Gamma \models^{\exists} a = b}\ \textsc{EEq}$$

$$\boxed{\Gamma \models^{\exists} \mathsf{injrng}\ A\ \mathsf{for}\ v}$$

$$\frac{\begin{array}{c}\forall A'B'.\, \Gamma \models^{\exists} ((x : A) \rightarrow B) = ((x : A') \rightarrow B')\\ \text{implies}\ \Gamma \models^{\exists} \{v/x\}\, B = \{v/x\}\, B'\end{array}}{\Gamma \models^{\exists} \mathsf{injrng}\ (x : A) \rightarrow B\ \mathsf{for}\ v}\ \textsc{EIRPi}$$

$$\frac{\begin{array}{c}\forall A'B'.\, \Gamma \models^{\exists} (\bullet(x : A) \rightarrow B) = (\bullet(x : A') \rightarrow B')\\ \text{implies}\ \Gamma \models^{\exists} \{v/x\}\, B = \{v/x\}\, B'\end{array}}{\Gamma \models^{\exists} \mathsf{injrng}\ (x : A) \rightarrow B\ \mathsf{for}\ v}\ \textsc{EIRIPi}$$

Figure 10. Bidirectional typing rules for surface language

$$\boxed{\Gamma \vDash \mathsf{injrng}\ A\ \mathsf{for}\ v}$$

$$\frac{\begin{array}{l}\Gamma \vdash v : A \quad \Gamma \vdash (x\!:\!A) \to B : \mathsf{Type} \\ \forall A'\,B'.((\Gamma \vDash ((x\!:\!A) \to B) = ((x\!:\!A') \to B')) \text{ implies } (\Gamma \vDash \{v/x\}\,B = \{v_{\triangleright v_0}/x\}\,B' \text{ where } \Gamma \vdash v_0 : A = A'))\end{array}}{\Gamma \vDash \mathsf{injrng}\ (x\!:\!A) \to B\ \mathsf{for}\ v}\ \text{IRPI}$$

$$\frac{\begin{array}{l}\Gamma \vdash v : A \quad \Gamma \vdash \bullet(x\!:\!A) \to B : \mathsf{Type} \\ \forall A'\,B'.((\Gamma \vDash (\bullet(x\!:\!A) \to B) = (\bullet(x\!:\!A') \to B')) \text{ implies } (\Gamma \vDash \{v/x\}\,B = \{v_{\triangleright v_0}/x\}\,B' \text{ where } \Gamma \vdash v_0 : A = A'))\end{array}}{\Gamma \vDash \mathsf{injrng}\ \bullet(x\!:\!A) \to B\ \mathsf{for}\ v}\ \text{IRIPI}$$

Figure 11. Core language injectivity restriction

It is also interesting to note the rules that do *not* appear in Figure 10. For example, there is no rule or surface syntax corresponding to TCAST, because this feature can be written as a user-level function. Similarly, the rather involved machinery for rewriting subterms and and erased terms (rule TJSUBST) can be entirely omitted, since it is subsumed by the congruence closure relation. The programmer only needs to introduce the equations into the context and they will be used automatically.

Finally we note that the surface language does not satisfy some of the usual properties of type systems. In particular, it lacks a general weakening lemma because the injrng relation cannot be weakened. Similarly, it does not satisfy a substitution lemma because that property fails for the congruence closure relation. (We might expect that $\Gamma, x : C \vDash a = b$ and $\Gamma \vdash v : C$ would imply $\Gamma \vDash \{v/x\}\,a = \{v/x\}\,b$. But this fails if C is an equation and the proof v makes use of the operational semantics.) And it does not satisfy a strengthening lemma, because even variables that do not occur in a term may be implicitly used as assumptions of congruence proofs.

The situations where weakening and substitution fail are rare (we have never encountered one when writing example programs in ZOMBIE) and there are straightforward workarounds for programmers. Furthermore, these properties do hold for fully annotated expressions, so there are no restrictions on the output of elaboration. However, the typing rules for the declarative system must be formulated to avoid these issues, which requires some extra premises. The rule IVAR requires $\Gamma \vdash A \Leftarrow \mathsf{Type}$ (proving this from $\vdash \Gamma \Leftarrow$ would need weakening); IAPP requires $\Gamma \vdash B \Leftarrow \mathsf{Type}$ (proving this from $\Gamma \vdash A \to B : \mathsf{Type}$ would need strengthening); and CREC requires $\Gamma, f : (x\!:\!A_1) \to A_2 \vdash (x\!:\!A_1) \to A_2 \Leftarrow \mathsf{Type}$ (proving this from $\Gamma \vdash (x : A_1) \to A_2 \Leftarrow \mathsf{Type}$ would need weakening).

6. Elaboration

We implement the declarative system using an elaborating typechecker, which translates a surface language expression (if it is well-formed according to the bidirectional rules) to an expression in the core language.

We formalize the algorithm that the elaborator uses as two inductively defined judgements, written $\Gamma' \mathrel{\scriptstyle\vdash} a \Rightarrow a' : A'$ (Γ' and a are inputs) and $\Gamma' \mathrel{\scriptstyle\vdash} a \Leftarrow A' \rightsquigarrow a'$ (Γ', a, and A' are inputs). The variables with primes (Γ', a' and A') are fully annotated expressions in the core language, while a is the surface language term being elaborated. The elaborator deals with each top-level definition in the program separately, and the context Γ' is an input containing the types of the previously elaborated definitions.

The job of the elaborator is to insert enough annotations in the term to create a well-typed core expression. It should not otherwise change the term. Stated more formally,

***Theorem* 4** (Elaboration soundness)**.**

1. If $\Gamma \mathrel{\scriptstyle\vdash} a \Rightarrow a' : A$, then $\Gamma \vdash a' : A$ and $|a| = |a'|$.
2. If $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma \mathrel{\scriptstyle\vdash} a \Leftarrow A \rightsquigarrow a'$, then $\Gamma \vdash a' : A$ and $|a| = |a'|$.

Furthermore, the elaborator should accept those terms specified by the declarative system. If the type system of Section 5 accepts a program, then the elaborator succeeds (and produces an equivalent type in inference mode).

***Theorem* 5** (Elaboration completeness)**.**

1. If $\Gamma \vdash a \Rightarrow A$ and $\mathrel{\scriptstyle\vdash} \Gamma \rightsquigarrow \Gamma'$ and $\Gamma' \mathrel{\scriptstyle\vdash} A \Leftarrow \mathsf{Type} \rightsquigarrow A'$, then $\Gamma' \mathrel{\scriptstyle\vdash} a \Rightarrow a' : A''$ and $\Gamma' \vDash A' = A''$
2. If $\Gamma \vdash a \Leftarrow A$ and and $\mathrel{\scriptstyle\vdash} \Gamma \rightsquigarrow \Gamma'$ and $\Gamma' \mathrel{\scriptstyle\vdash} A \Leftarrow \mathsf{Type} \rightsquigarrow A'$, then $\Gamma' \mathrel{\scriptstyle\vdash} a \Leftarrow A' \rightsquigarrow a'$.

Designing the elaboration rules follows the standard pattern of turning a declarative specification into an algorithm: remove all rules that are not syntax directed (in this case ICAST and CCAST), and generalize the premises of the remaining rules to create a syntax-directed system that accepts the same terms. At the same time, the uses of congruence closure relation $\Gamma \vDash a = b$, must be replaced by appropriate calls to the congruence closure algorithm. We specify this algorithm using the following (partial) functions:

$\Gamma \mathrel{\scriptstyle\vdash} A \stackrel{?}{=} B \rightsquigarrow v$, which checks A and B for equality and produces core-language proof v.
$\Gamma \mathrel{\scriptstyle\vdash} A =^{?} (x : B_1) \to B_2 \rightsquigarrow v$, which checks whether A is equal to some function type and produces that type and proof v.
$\Gamma \mathrel{\scriptstyle\vdash} A =^{?} (B_1 = B_2) \rightsquigarrow v$, which is similar to above, except for equality types.

For example, consider the rule for elaborating function applications:

$$\frac{\begin{array}{l}\Gamma \mathrel{\scriptstyle\vdash} a \Rightarrow a' : A_1 \quad \Gamma \mathrel{\scriptstyle\vdash} A_1 =^{?} (x : A) \to B \rightsquigarrow v_1 \\ \Gamma \mathrel{\scriptstyle\vdash} v \Leftarrow A \rightsquigarrow v' \quad \Gamma \vDash \mathsf{injrng}\ (x\!:\!A) \to B\ \mathsf{for}\ v'\end{array}}{\Gamma \mathrel{\scriptstyle\vdash} a\ v \Rightarrow a'_{\triangleright v_1}\ v' : \{v'/x\}\,B}\ \text{EIDAPP}$$

In the corresponding declarative rule (IDAPP) the applied term a must have an arrow type, but this can be arranged by implicitly using ICAST to adjust a's type. Therefore, in the algorithmic system, the corresponding condition is that the type of a should be equal to an arrow type $(x : A) \to B$ modulo the congruence closure. Operationally, the typechecker will infer some type A_1 for a, then run the congruence closure algorithm to construct the set of all expressions that are equal to A_1, and check if the set contains some expression which is an arrow type. The elaborated core term uses the produced proof of $A_1 = (x:A) \to B$ in a cast to change the type of a.

At this point there is a potential problem: what if A_1 is equal to more than one arrow type? For example, if $A_1 = (x:A) \to B = (x:A') \to B$, then the elaborator has to choose whether to check b against A or A'. A priori it is quite possible that only one of them will work; for example the context Γ may contain an inconsistent equation like $\mathsf{Nat} \to \mathsf{Nat} = \mathsf{Bool} \to \mathsf{Nat}$. We do not wish to introduce a backtracking search here, because that could make type checking too slow.

This kind of mismatch in the domain type can be handled by extending the congruence closure algorithm. Note that things are fine if $\Gamma \vDash A = A'$, since then it does not matter if A or A' is chosen. So the issue only arises if $\Gamma \vDash (x:A) \to B = (x:A') \to B$ and not $\Gamma \vDash A = A'$. Fortunately, type constructors are injective in the core language (Section 3). Including injectivity as part of the congruence closure judgement (by the rule TCCINJDOM) ensures that it does not matter which arrow type is picked.

We also have to worry about a mismatch in the codomain type, i.e. the case when $\Gamma \vDash A_1 = (x:A) \to B$ and $\Gamma \vDash A_1 = (x:A') \to B'$ for two different types. At first glance it seems as if we could use the same solution. After all, the core language includes a rule for injectivity of the range of function types (rule TJINJRNG). There is an important difference between this rule and TJINJDOM, however, which is the handling of the bound variable x in the codomain B: the rule says that this can be closed by substituting any value for it. As a result, we cannot match this rule in the congruence closure relation, because the algorithm would have to guess that value. In other words, to match this rule in the congruence closure relation would mean to add a rule like

$$\frac{\Gamma \vDash (x:A) \to B = (x:A) \to B' \qquad \Gamma \vdash v : A}{\Gamma \vDash \{v/x\}\, B = \{v/x\}\, B'}$$

This proposed rule is an axiom schema, which can be instantiated for any value v. Unfortunately, that makes the resulting equational theory undecidable.

For example, the equational theory of SKI-combinators (which is known undecidable) could be encoded as an assumption context containing one indexed datatype `T` and two equations:

```
data SK = S | K | App of SK SK

T : SK → Type
ax1 : ((x y : SK) →
    T (App (App K x) y)) = ((x y : SK) → T x)
ax2 : ((f g x : SK) →
    T (App (App (App S f) g) x)) = ...
```

As far as writing an elaborator goes, maybe this is fine—after all, we only want to apply the axiom to the particular value v from the function application $a\ v$. However, there does not seem to be any natural way to write a declarative specification explaining what values v should be candidates.

Instead, we restrict the declarative language to forbid this problematic case. That is, the programmer is not allowed to write a function application unless all possible return types for the function are equal. Note that in cases when an application is forbidden by this check, the programmer can avoid the problem by proving the required equation manually and ensuring that it is available in the context.

In the fully-annotated core language we express this restriction with the rule IRPI (in Figure 11), and then lift this operation to partially annotated terms by rule EIRPI (Figure 10). Operationally, the typechecker will search for all arrow types equal to A_1 and check that the the codomains with v substituted are equal in the congruence closure. This takes advantage of the fact that equivalence classes under congruence closure can be efficiently represented—although the rule as written appears to quantify over potentially infinitely many function types, the algorithm in Section 7 will represent these as a finite union-find structure which can be effectively enumerated. In the core language rule we need to insert a type coercion from A to A' to make the right-hand side well typed. By the rule TCCINJDOM that equality is always provable, so the typechecker will use the proof term v_0 that the congruence closure algorithm produced.

In the case of a simple arrow type $A \to B$, the range injectivity rule is unproblematic and we do include it in the congruence closure relation (TCCINJRNG). So the application rule for simply-typed functions (EIAPP) does not need the injectivity restriction. On the other hand, if the core language did not support injectivity for arrow domains, we could have used the same injectivity restriction for both the domain and codomain.

The rule for checking function definitions (ECREC)

$$\frac{\begin{array}{l}\Gamma \Vdash A =^? (x:A_1) \to A_2 \rightsquigarrow v_1 \\ \Gamma, f:(x:A_1) \to A_2, x:A_1 \Vdash a \Leftarrow A_2 \rightsquigarrow a' \\ \Gamma, f:(x:A_1) \to A_2, x:A_1 \vDash \mathsf{injrng}\,(x:A_1) \to A_2 \text{ for } x \\ \Gamma, f:(x:A_1) \to A_2 \Vdash (x:A_1) \to A_2 \Leftarrow \mathsf{Type} \rightsquigarrow A_0\end{array}}{\Gamma \Vdash \mathsf{rec}\ f\ x.a \Leftarrow A \rightsquigarrow (\mathsf{rec}\ f_{(x A_1) \to A_2}\ x.a')_{\triangleright \mathsf{symm}\ v_1}}\ \text{ECREC}$$

uses the same ideas that we saw in the application rule. First, while the declarative rule checks against a syntactic arrow type, the algorithmic system searches whether the type A is equivalent to some arrow type $(x:A_1) \to A_2$. Second, to avoid trouble if there is more than one such function type, we add an injrng restriction.

Thus the ECREC rule ensures that although there may be some choice about what type A_1 to give to the new variable x in the context, all the types that can be chosen are equal up to CC. We then need to design the type system so that all judgements are invariant under-CC equivalent contexts.

The rest of the elaborations rules hold few surprises. The rules for computationally irrelevant abstractions and applications (EIIPI, EIIDAPP, and ECIREC) are exactly analog to the rules for relevant functions.

On the checking side, the mode-change rule ECINF now needs to prove that the synthesized and checked types are equal.

$$\frac{\Gamma \Vdash a \Rightarrow a' : A \quad \Gamma \Vdash A \stackrel{?}{=} B \rightsquigarrow v_1}{\Gamma \Vdash a \Leftarrow B \rightsquigarrow a'_{\triangleright v_1}}\ \text{ECINF}$$

This rule corresponds to a direct call to the congruence closure algorithm, producing a proof term v_1. Note that the inputs are fully elaborated terms—in moving from the declarative to the algorithmic type system, we replaced the undecidable condition $\Gamma \vDash^{\exists} A = B$ with a decidable one.

Finally, the rule ECREFL elaborates checkable equality proofs (written as underscores in the concrete ZOMBIE syntax).

$$\frac{\Gamma \Vdash A =^? (a = b) \rightsquigarrow v_1 \quad \Gamma \Vdash a \stackrel{?}{=} b \rightsquigarrow v}{\Gamma \Vdash \mathsf{join} \Leftarrow A \rightsquigarrow v_{\triangleright \mathsf{symm}\ v_1}}\ \text{ECREFL}$$

As in the rule for application, the typechecker does a search through the equivalence class of the ascribed type A to see if it contains any equations. If there is more than one equation it does not matter which one gets picked, because the congruence relation includes injectivity of the equality type constructor (TCCINJEQ). In the elaborated term we need to prove $(a = b) = A$ given $A = (a = b)$. This can be done using TJOINP (for reflexivity) and TJSUBST, and we abbreviate that proof term $\mathsf{symm}\ v_1$.

6.1 Properties of the congruence closure algorithm

It is attractive to base our type system around congruence closure because there exists efficient algorithms to decide it. But the correctness proof for the elaborator does not need to go into details about how that algorithm work. It only assumes that the congruence closure algorithm satisfies the following properties. (We show the statement of these properties for function types below, the others are similar.)

***Property* 6** (Soundness)**.** If $\Gamma \vDash A \stackrel{?}{=} (x : B_1) \to B_2 \rightsquigarrow v$, then $\Gamma \vdash v : A = ((x : B_1) \to B_2)$ and $|v| = \mathsf{join}$ and $\Gamma \vDash A = (x\!:\!B_1) \to B_2$.

***Property* 7** (Completeness)**.** If $\Gamma \vDash A = (x\!:\!B_1) \to B_2$ then there exists a B_1', B_2' and v such that $\Gamma \vDash A \stackrel{?}{=} (x\!:\!B_1') \to B_2' \rightsquigarrow v$.

***Property* 8** (Respects Congruence Closure)**.** If $\Gamma \vDash A = B$ and $\Gamma \vDash B \stackrel{?}{=} (x\!:\!C_1) \to C_2 \rightsquigarrow v$ then $\Gamma \vDash A \stackrel{?}{=} ((x\!:\!C_1') \to C_2') \rightsquigarrow v'$.

In other words, the algorithm should be sound and complete with respect to the $\Gamma \vDash A = B$ relation; it should generate correct core proof terms v; and the output should depend only on the equivalence class the input is in. In the next section we show how to implement an algorithm satisfying this interface.

7. Implementing congruence closure

Algorithms for congruence closure in the first-order setting are well studied, and our work builds on them. However, in our type system the relation $\Gamma \vDash a = b$ does more work than "classic" congruence closure: we must also handle erasure, terms with bound variables, (dependently) typed terms, the injectivity rules, the "assumption up to congruence" rule, and we must generate proof terms in the core language.

Our implementation proves an equation $a = b$ in three steps. First, we erase all annotations from the input terms and explicitly mark places where the congruence rule can be applied, using an operation called *labelling*. Then we use an adapted version of the congruence closure algorithm by Nieuwenhuis and Oliveras [23]. Our version of their algorithm has been extended to also handle injectivity and "assumption up to congruence", but it ignores all the checks that the terms involved are well-typed. Finally, we take the untyped proof of equality, and process it into a proof that a and b are also related by the typed relation. The implementation is factored in this way because the congruence rule does not necessarily preserve well-typedness, so the invariants of the algorithm are easier to maintain if we do not have to track well-typedness at the same time.

7.1 Labelling terms

In $\Gamma \vDash a = b$, the rule TCCCONGRUENCE is stated in terms of substitution. But existing algorithms expect congruence to be applied only to syntactic function applications: from $a = b$ conclude $f\ a = f\ b$. To bridge this gap, we preprocess equations into (erased) *labelled expressions*. A label F is an erased language expression with some designated holes (written –) in it, and a labelled expression is a label applied to zero or more labelled expressions, i.e. a term in the following grammar.

$$a ::= F\,\overline{a_i}$$

Typically a label will represent just a single node of the abstract syntax tree. For example, a wanted equation $f\ x = f\ y$ will be processed into $(–\ –)\ f\ x = (–\ –)\ f\ y$, where the label $(–\ –)$ means this is an application. However, for syntactic forms involving bound variables, it can be necessary to be more coarse-grained. For example, given $a = b$ our implementation can prove $\mathsf{rec}\ f\ x.a+x = \mathsf{rec}\ f\ x.b+x$, which involves using $\mathsf{rec}\ f\ x.–+x$ as a label. In general, to process an expression a into a labelled term, the implementation will select the largest subexpressions that do not involve any bound variables.

The labelling step also deletes all annotations from the input expressions. This means that we automatically compute the congruence closure up to erasure (rule TCCERASURE), at the cost of needing to do more work when we generate core language witnesses (Section 7.3).

Applying the labelling step simplifies the congruence closure problems in several ways. We show the simpler problem by defining the relation $\Gamma \vDash^{\mathsf{L}} a = b$ defined in Figure 12. Compared to Figure 9 we no longer need a rule for erasure, congruence is only used on syntactic label applications, all the different injectivity rules are handled generically, and we do not ensure that the terms are well-typed. In the appendix we formally define the label operation, and prove that it is complete in the following sense.

***Lemma* 9.** If $\Gamma \vDash a = b$ then $\mathsf{label}\,\Gamma \vDash^{\mathsf{L}} \mathsf{label}\ a = \mathsf{label}\ b$.

7.2 Untyped congruence closure

Next, we use an algorithm based on Nieuwenhuis and Oliveras [23] to decide the $\Gamma \vDash^{\mathsf{L}} a = b$ relation. The algorithm first "flattens" the problem by allocating *constants* c_i (i.e. fresh names) for every subexpression in the input. After this transformation every input equation has either the form $c_1 = c_2$ or $c = F(c_1, c_2)$, that is, it is either an equation between two atomic constants or between a constant and a label F applied to constants. Then follows the main loop of the algorithm, which is centered around three data-structures: a queue of input equations, a union-find structure and a lookup table. In each step of the loop, we take off an equation from the queue and update the state accordingly. When all the equations have been processed the union-find structure represents the congruence closure.

The union-find structure tracks which constants are known to be equal to each other. When the algorithm sees an input equation $c_1 = c_2$ it merges the corresponding union-find classes. This deals with the reflexivity, symmetry and transitivity rules. The lookup table is used to handle the congruence rule. It maps applications $F(c_1, c_2)$ to some canonical representative c. If the algorithm sees an input equation $c = F(c_1, c_2)$, then c is recorded as the representative. If the table already had an entry c', then we deduce a new equation $c = c'$ which is added to the queue.

In order to adapt this algorithm to our setting, we make three changes. First, we adapt the lookup tables to include the *richer labels* corresponding to the many syntactic categories of our core language. (Nieuwenhuis and Oliveras only use a single label meaning "application of a unary function.")

Second, we deal with *injectivity rules* in a way similar to the implementation of Coq's `congruence` tactic [12]. Certain labels are considered injective, and in each union-find class we identify

$$\frac{}{\Gamma \vdash^{\mathsf{L}} a = a}\textsc{LCCRefl} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a = b}{\Gamma \vdash^{\mathsf{L}} b = a}\textsc{LCCSym} \qquad \frac{\Gamma \vdash^{\mathsf{L}} a_1 = b \quad \Gamma \vdash^{\mathsf{L}} b = a_2}{\Gamma \vdash^{\mathsf{L}} a_1 = a_2}\textsc{LCCTrans}$$

$$\frac{x : A \in \Gamma \quad \Gamma \vdash^{\mathsf{L}} A = ((- = -)\ a\ b)}{\Gamma \vdash^{\mathsf{L}} a = b}\textsc{LCCAssum} \qquad \frac{\forall k.\ \Gamma \vdash^{\mathsf{L}} a_k = b_k}{\Gamma \vdash^{\mathsf{L}} F\,\overline{a_i} = F\,\overline{b_i}}\textsc{LCCCong} \qquad \frac{\Gamma \vdash^{\mathsf{L}} F\ a = F\ b \quad F\ \mathsf{injective}}{\Gamma \vdash^{\mathsf{L}} a = b}\textsc{LCCInj}$$

Figure 12. Untyped congruence closure on labelled terms

the set of terms that start with an injective label. If we see an input equation $c = F(c_1, c_2)$ and F is injective we record this in the class of c. Whenever we merge two classes, we check for terms headed by the same F; e.g. if we merge a class containing $F(c_1, c_2)$ with a class containing $F(c_1', c_2')$, we deduce new equations $c_1 = c_1'$ and $c_2 = c_2'$ and add those to the queue.

Third, our implementation of the *extended assumption rule* works much like injectivity. With each union-find class we record two new pieces of information: whether any of the constants in the class (which represent types of our language) are known to be inhabited by a variable, and whether any of the constants in the class represents an equality type. Whenever we merge two classes we check for new equations to be added to the queue.

In Appendix D we give a precise description of our algorithm, and prove its correctness, i.e. that it terminates and returns "yes" iff the wanted equation is in the $\Gamma \vdash^{\mathsf{L}} a = b$ relation.

First we prove that flattening a context does not change which expressions are equal in that context. Although the flattening algorithm itself is the same as in previous work, the statement of the correctness proof is refined to say that the new assumptions h are always used as plain assumptions $h_{\triangleright \mathsf{refl}}$, as opposed to the general assumption-up-to-CC rule $h_{\triangleright p}$. The distinction is important, because although the flattening algorithm will process every assumption that was in the original context, it does not go on to recursively flatten the new assumptions that it added. So for completeness of the whole algorithm we need to know that there is never a need to reason about equality between such assumptions.

Then the correctness proof of the main algorithm is done in two parts. The *soundness* of the algorithm (i.e. if the algorithm says "yes" then the two terms really are provably equal) is fairly straightforward. We verify the invariant that every equation which is added to the input queue, union-find structure, and lookup table really is provably true. For each step of the algorithm which extends these datastructures we check that the new equation is provable from the already known ones. In fact, this proof closely mirrors the way the implementation in ZOMBIE works: there the datastructures contain not only bare equations but also the evidence terms that justify them (see section 7.3), and each step of the algorithm builds new evidence terms from existing ones.

The *completeness* direction (if $\Gamma \vdash^{\mathsf{L}} a = b$ then the algorithm will return "yes") is more involved. We need to prove that at the end of a run of the algorithm, the union-find structure satisfies all the proof rules of the congruence relation. For our injectivity rule and extended assumption rule this means properties like

- For all $\overline{a_i}$ and $\overline{b_i}$, if $F\,\overline{a_i} \approx_R F\,\overline{b_i}$ and F is injective, then $\forall k. a_k \approx_R b_k$.
- If $x : A \in \Gamma$ then for all a, b, if $A \approx_R (a = b)$ then $a \approx_R b$.

where $\approx_R$ denotes the equivalence relation generated by the union-find links. The proof uses a generalized invariant: while the algorithm is still running R satisfies the proof rules modulo the pending equations E in the input queue, e.g. the invariant for the assumption rule is

> If $x : A \in \Gamma$ then for all a, b, if $A \approx_R (a = b)$ then $a \approx_{E,R} b$.

However, the congruence rule presents some extra difficulties. The full congruence relation for a given context Γ is in general infinite (if $\Gamma \vdash^{\mathsf{L}} a = b$, then by the congruence rule we will also have $\Gamma \vdash^{\mathsf{L}} \mathsf{S}\ a = \mathsf{S}\ b$ and $\Gamma \vdash^{\mathsf{L}} \mathsf{S}\ (\mathsf{S}\ a) = \mathsf{S}\ (\mathsf{S}\ b)$ and ...). So at the end of the run of an algorithm the datastructures will not contain information about all possible congruence instances, but only those instances that involve terms from the input problem.

Following Corbineau [11] we attack this problem in two steps. First we show that at the end of the run of the algorithm the union-find structure R *locally* satisfies the congruence rule in the following sense:

- If $a_i \approx_R b_i$ for all $0 \leq i < n$, and $F\,\overline{a_i}$ and $F\,\overline{b_i}$ both appeared in the list of input equations, then $F\,\overline{a_i} \approx_R F\,\overline{b_i}$.

We then need to prove that this local completeness implies completeness. This amounts to showing that if a given statement $\Gamma \vdash^{\mathsf{L}} a = b$ is provable at all, it is provable by using the congruence rule only to prove equations between subexpressions of Γ, a, and b. There are a few approaches to this in the literature. The algorithm by Nieuwenhuis and Oliveras [23] can be shown correct because it is an instance of Abstract Congruence Closure (ACC) [4], while the correctness proofs for ACC algorithms in general relies on results from rewriting theory. However, it is not immediately obvious how to generalize this approach to handle additional rules like injectivity. Corbineau [11] instead gives a semantic argument about finite and general models.

As it happens, in our development there is a separate reason for us to prove that local uses of the congruence rule suffice: we need this result to bridge the gap between untyped and typed congruence. This is the subject of Section 7.3, and we use the lemmas from that section to finish the completeness argument. All in all, this yields:

***Lemma* 10.** The algorithm described above is a decision procedure for the relation $\Gamma \vdash^{\mathsf{L}} a = b$.

7.3 Typing restrictions and generating core language proofs

Along the pointers in the union-find structure, we also keep track of the evidence that showed that two expressions are equal. The syntax of the evidence terms is given by the following grammar. An evidence term p is either an assumption x (with a proof p that x's type is an equation), reflexivity, symmetry, transitivity, injectivity, or an application of congruence annotated with a label A.

$$p, q \quad ::= \quad x_{\triangleright p} \mid \mathsf{refl} \mid p^{-1} \mid p; q \mid \mathsf{inj}_i\ p \mid \mathsf{cong}_{A}\ p_1 \mathinner{..} p_i$$

Next we need to turn the evidence terms p into proof terms in the core calculus. This is nontrivial, because the Nieuwenhuis-Oliveras algorithm does not track types. Not every equation which is derivable by untyped congruence closure is derivable in the typed theory; for example, if $f : \mathsf{Bool} \to \mathsf{Bool}$, then from the equation $(a : \mathsf{Nat}) = (b : \mathsf{Nat})$ we can not conclude $f\ a = f\ b$, because $f\ a$ is not a well-typed term. Worse still, even if the conclusion is well-typed, not every untyped *proof* is valid in the typed theory, because it may involve ill-typed intermediate terms. For example, let $\mathsf{Id} : (A : \mathsf{Type}) \to A \to A$ be the polymorphic identity function, and suppose we have some terms $a : A$, $b : B$, and know the equations $x : A = B$ and $y : a = b$. Then

$$(\mathsf{cong}_{\mathsf{Id}}\ x\ \mathsf{refl}); (\mathsf{cong}_{\mathsf{Id}}\ \mathsf{refl}\ y)$$

is a valid untyped proof of $\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ b$. But it is not a correct typed proof because it involves the ill-typed term $\mathsf{Id}\ B\ a$:

$$\dfrac{\dfrac{x : A = B \qquad a = a}{\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ a}\ \mathit{cong} \qquad \dfrac{B = B \qquad y : a = b}{\mathsf{Id}\ B\ a = \mathsf{Id}\ B\ b}\ \mathit{cong}}{\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ b}\ \mathit{trans}$$

Corbineau [12] notes this as an open problem. Of course, the above proof is unnecessarily complicated. The same equation can be proved by a single use of congruence.

$$\dfrac{x : A = B \qquad y : a = b}{\mathsf{Id}\ A\ a = \mathsf{Id}\ B\ b}\ \mathit{cong}$$

Furthermore, the simpler proof does not have any issues with typing: every expression occurring in the derivation is either a subexpression of the goal or a subexpression of one of the equations from the context, so we know they are well-typed.

Our key observation is that this trick works in general. The only time a congruence proof will involve expressions which were not already present in the context or goal is when transitivity is applied to two derivations ending in cong. We simplify such situations using the following CONGTRANS rule.

$$(\mathsf{cong}_{\,A}\ p_1 \mathinner{..} p_i); (\mathsf{cong}_{\,A}\ q_1 \mathinner{..} q_i) \mapsto \mathsf{cong}_{\,A}\ (p_1; q_1) \mathinner{..} (p_i; q_i)$$

This rule is valid in general, and it does not make the proof larger. We also need rules for simplifying evidence terms that combine transitivity with injectivity or assumption-up-to-CC, such as $\mathsf{inj}_i\ (\mathsf{cong}_{\,A}\ p_1 \mathinner{..} p_k)$ and $x_{\triangleright(r;\mathsf{cong}_{=}\ p\ q)}$, rules for pushing uses of symmetry $(^{-1})$ past the other evidence constructors, and rules for rewriting subterms. The complete simplification relation $\mapsto$ is shown in Figure 13.

Any evidence term p can be simplified into a normalized evidence term p^*. (In the appendix we define an explicit grammar for fully simplified terms p^*, and prove than any term can be simplified into that form). And given p^* it is easy to produce a corresponding proof term in the core language. The idea is that one can reconstruct the middle expression in every use of transitivity $(p; q)$, because at least one of p and q will be specific enough to pin down exactly what equation it is proving. Formally, we define the judgement $\Gamma \vdash^{\mathsf{L}} p : a = b$ by adding evidence terms to the rules in Figure 12, and then prove:

***Lemma* 11.** If we have $\mathsf{label}\,\Gamma \vdash^{\mathsf{L}} p^* : \mathsf{label}\,a = \mathsf{label}\,b$ and $\Gamma \vdash a = b : \mathsf{Type}$, then $\Gamma \vDash a = b$.

Simplifying the evidence terms also solves another issue, which arises because of the TCCERASURE rule. Because the input terms are preprocessed to delete annotations (Section 7.1), an arbitrary evidence term will not uniquely specify the annotations. For example, change the previous example by making the type parameter an *erased* argument of Id, and suppose we have assumptions $x : a = a'$ and $y : a' = b$. Then the evidence term $(\mathsf{cong}_{\,\mathsf{Id}\ \bullet\ \_}\ x); (\mathsf{cong}_{\,\mathsf{Id}\ \bullet\ \_}\ y)$ could serve as the skeleton of either the valid proof

$$\dfrac{\dfrac{x : a = a'}{\mathsf{Id}\ \bullet_A\ \ a = \mathsf{Id}\ \bullet_A\ \ a'}\ \mathit{cong} \qquad \dfrac{y : a' = b}{\mathsf{Id}\ \bullet_A\ \ a' = \mathsf{Id}\ \bullet_B\ \ b}\ \mathit{cong}}{\mathsf{Id}\ \bullet_A\ \ a = \mathsf{Id}\ \bullet_B\ \ B}\ \mathit{trans}$$

or the invalid proof

$$\dfrac{\dfrac{x : a = a'}{\mathsf{Id}\ \bullet_A\ \ a = \mathsf{Id}\ \bullet_B\ \ a'}\ \mathit{cong} \qquad \dfrac{y : a' = b}{\mathsf{Id}\ \bullet_B\ \ a' = \mathsf{Id}\ \bullet_B\ \ b}\ \mathit{cong}}{\mathsf{Id}\ \bullet_A\ \ a = \mathsf{Id}\ \bullet_B\ \ B}\ \mathit{trans}$$

Again, this issue only arises because of the cong-trans pair. Simplifying the evidence term resolves the issue, because in a simplified term every intermediate expression is pinned down.

Putting together the labelling step, the evidence simplification step and the proof term generation step we can relate typed and untyped congruence closure. In the following theorem, the relation $\Gamma \vdash a = b$ is defined by similar rules as Figure 9 except that we omit the typing premises in TCCREFL, TTCERASURE and TTCCONGRUENCE.

***Theorem* 12.** Suppose $\Gamma \vdash a = b$ and $\Gamma \vdash a = b : \mathsf{Type}$. Then $\Gamma \vDash a = b$. Furthermore $\Gamma \vdash v : a = b$ for some v.

The computational content of the proof is how the elaborator generates core language evidence for equalities, so this shows the correctness of the ZOMBIE implementation. But it is also interesting as a theoretical result in its own right, and an important part of the proof of completeness of elaboration (Section 6).

8. Extensions

The full ZOMBIE implementation includes more features than the surface language described in Section 5. We omitted them from the formal system in order to simplify the proofs, but they are important to make programming up to congruence work well.

8.1 Smart case

Although we do not include datatypes in this paper, they are a part of the ZOMBIE implementation, and an important component of any dependently-typed language. The presence of congruence closure elaboration means that the core language [28] can use a specification of dependently-typed pattern matching called *smart case* [1].

With smart case, the rule for case analysis introduces a new equation into the context when checking each branch of a case expression. For example, the rule for an if expression type checks each branch under the assumption that the condition is true or false.

$$\dfrac{\begin{array}{l}\Gamma \vdash a : \mathsf{Bool}\\ \Gamma, x : a = \mathsf{true} \vdash b_1 : A\\ \Gamma, x : a = \mathsf{false} \vdash b_2 : A\end{array}}{\Gamma \vdash \mathsf{if}\ a\ \mathsf{then}\ b_1\ \mathsf{else}\ b_2 : A}\ \textsc{TFullCase}$$

This rule is in contrast to specifications that use *unification* to communicate the information gained by pattern matching. In those systems, if the scrutinee and the patterns are not unifiable (in the fragment of higher-order unification supported by the type system) then the case expression must be rejected. Furthermore, the specification of the typing rule for the unification based systems is more complicated. Smart case, by pushing this information to propositional equality, is both simpler and more expressive.

$$
\begin{array}{lcl}
p & \mapsto & p \\
\mathsf{refl}^{-1} & \mapsto & \mathsf{refl} \\
\mathsf{refl}; p & \mapsto & p \\
p; \mathsf{refl} & \mapsto & p \\
(p; q); r & \mapsto & p; (q; r) \\
p; p^{-1} & \mapsto & \mathsf{refl} \\
p^{-1}; p & \mapsto & \mathsf{refl} \\
 & & \\
{p^{-1}}^{-1} & \mapsto & p \\
(p; q)^{-1} & \mapsto & q^{-1}; p^{-1}
\end{array}
\qquad
\begin{array}{rcl}
(\mathsf{cong}\ _A\, p_1 \mathrel{..} p_i)^{-1} & \mapsto & \mathsf{cong}\ _A\, {p_1}^{-1} \mathrel{..} {p_i}^{-1} \\
 & & \\
(\mathsf{inj}_i\ p)^{-1} & \mapsto & \mathsf{inj}_i\ (p^{-1}) \\
 & & \\
(\mathsf{cong}\ _A\, p_1 \ldots p_i); (\mathsf{cong}\ _A\, q_1 \ldots q_i) & \mapsto & \mathsf{cong}\ _A\, (p_1; q_1) \mathrel{..} (p_i; q_i) \\
 & & \\
\mathsf{inj}_k\ (\mathsf{cong}\ _A\, p_1 \ldots p_i) & \mapsto & p_k \\
\mathsf{inj}_k\ ((\mathsf{cong}\ _A\, p_1 \mathrel{..} p_i); r) & \mapsto & p_k; (\mathsf{inj}_k\ r) \\
\mathsf{inj}_k\ (r; (\mathsf{cong}\ _A\, p_1 \ldots p_i)) & \mapsto & (\mathsf{inj}_k\ r); p_k \\
x_{\triangleright (r; \mathsf{cong}\ _=\ p\ q)} & \mapsto & p^{-1}; (x_{\triangleright r}); q
\end{array}
$$

$$
\frac{p \mapsto p'}{x_{\triangleright p} \mapsto x_{\triangleright p'}}\ \textsc{Assumption}
\qquad
\frac{p \mapsto p' \quad q \mapsto q'}{p; q \mapsto p'; q'}\ \textsc{Trans}
\qquad
\frac{\forall k.\ \ p_k \mapsto p'_k}{\mathsf{cong}\ _A\, p_1 \mathrel{..} p_i \mapsto \mathsf{cong}\ _A\, p'_1 \mathrel{..} p'_i}\ \textsc{Cong}
\qquad
\frac{p \mapsto p'}{\mathsf{inj}_k\ p \mapsto \mathsf{inj}_k\ p'}\ \textsc{Inj}
$$

Figure 13. Simplification rules for evidence terms

The downside to smart case has been that because this information is recorded as an assumption in the context, it is more work for the programmer. However, with congruence closure, the type system is immediately able to take advantage of these equalities in each branch. Thus, the ZOMBIE surface language has the convenience of the unification-based rule, while the core language enjoys the simplicity of smart case.

8.2 Reduction modulo congruence

In the paper all β-reductions are introduced by expressions `join : a = b`. But in practice some additional support from the typechecker for common patterns can make programming much more pleasant.

First, one often wants to evaluate some expression a "as far as it goes". Then making the programmer write both sides of the equation $a = b$ is unnecessarily verbose. Instead we provide the syntax `unfold a in body`. The implementation reduces a to normal form, $a \rightsquigarrow_{\mathsf{cbv}} a' \rightsquigarrow_{\mathsf{cbv}} a'' \rightsquigarrow_{\mathsf{cbv}} a'''$ (if a does not terminate the programmer can specify a maximum number of steps), and then introduces the corresponding equations into the context with fresh names. That is, it elaborates as

```
let _ = (join : a = a') in
let _ = (join : a' = a'') in
let _ = (join : a'' = a''') in
  body
```

Second, many proofs requires an interleaving of evaluation and equations from the context, particularly in order to take advantage of equations introduced by smart case. One example is `npluszero` in Section 2. The case-expression needs to return a proof of `n+0 = n`. If we try to directly evaluate `n+0`, we would reach the stuck expression `case n of Zero → 0; Succ m' → Succ (m' + 0)`, so instead we used an explicit type annotation in the `Zero` branch to evaluate `0+0`. However, the context contains the equation `n = Zero`, which suggests that there should be another way to make progress.

To take advantage of such equations, we add some extra intelligence to the way unfold handles CBV-evaluation contexts, that is expressions of the form $f\ a$ or $(\mathsf{case}\ b\ \mathsf{of} \ldots)$. When encountering such an expression it will first recursively unfold the function f, the argument a, or the scrutinee b (as with ordinary CBV-evaluation), and add the resulting equations to the context. However, it will then examine the congruence equivalence class of these expressions to see if they contain any suitable values—any value v is suitable for a, a function value $\mathsf{rec}\ f\ \ x.a_0$ for f, and a value headed by a data constructor for b—and then unfold the resulting expression $(\mathsf{rec}\ f\ x.a_0)\ v$. (If there are several suitable values, one is selected arbitrarily). This way unfolding can make progress where ordinary CBV-evaluation gets stuck.

Using the same machinery we also provide a "smarter" version of `join`, which first unfolds both sides of the equation, and then checks that the resulting expressions are CC-equivalent. This lets us omit the type annotations from `npluszero`:

```
npluszero n = case n [eq] of
                Zero → smartjoin
                Suc m → ...
```

The unfold algorithm does not fully respect CC-equivalence, because it only converts *into* values. For example, suppose the context contains the equation $f\ \ a = v$, Then $\mathsf{unfold}\ g\ \ (f\ \ a)$ will evaluate $f\ \ a$ and add the corresponding equations to the context, but $\mathsf{unfold}\ g\ \ v$ will not cause $f\ \ a$ to be evaluated. This gives the programmer more control over what expressions are run.

We have not studied the theory of the unfold algorithm, and indeed it is not a complete decision procedure for our propositional equality. If a subexpression of a does not terminate, unfold will spend all its reduction budget on just that subexpression (but this is OK, because the programmer decides what expression a to unfold). And if the context contains e.g. an equation between two unrelated function values, unfold will arbitrarily choose one of them (but it is hard to think of an example where this would happen). We have found unfold very helpful when writing examples.

9. Related work

The annotated core language in this paper is a slight variation on previous work [28], which in turn is a subset of the full language implemented by ZOMBIE [10]. In this version, in order to keep the formalism small we omit some features (uncatchable exceptions and general datatypes) and replace the application rule with a slightly less expressive value-dependent version. However, these omissions are not significant (the original system is still compatible with the "up to congruence" approach and is implemented in ZOMBIE). We also took the opportunity to simplify some typing rules, and to emphasize the role of erasable annotations. Compared to the previous version, we replaced the old rule TCONV with two rules TCONG and TCAST, and we changed the rules for recursive functions.

The dependent application rule of the original system (and the one implemented in ZOMBIE) does not restrict its argument to be a value. Instead, this rule includes a premise that requires that the substituted type is well-formed. (With the value-restricted rule it is always well-formed, because the type system enjoys substitution for values, lemma 2). Thus the app rule looks as follows:

$$\frac{\begin{array}{l}\Gamma \vdash a : (x : A) \to B \\ \Gamma \vdash b : A \\ \Gamma \vdash \{b/x\}\, B : \mathsf{Type}\end{array}}{\Gamma \vdash a\ b : \{b/x\}\, B}\ \text{TFULLAPP}$$

When designing the elaborator, the premise $\Gamma \vdash \{b/x\}\, B : \mathsf{Type}$ requires attention. Among the arrow types that are equal to the type of the applied functions, there may be some where the resulting type $\{b/x\}\, B$ is well-formed and others where it is not. Because the congruence closure relation only equates well-typed expressions, the current definition of the $\Gamma \vDash \mathsf{injrng}\ A\ \mathsf{for}\ v$ says that the application is only allowed if *all* possible function types would lead to a well-formed result, and this check is what the current ZOMBIE implementation does. Perhaps one could instead search for *some* type which works—usually B will be a small expression, so the check for well-formedness can be done quickly. On the other hand, the question is somewhat academic, because in our experience the injrng condition is always satisfied in practice.

Propositional Equality The idea of using congruence closure is not limited to the particular version of propositional equality used by our core language, which has some nonstandard features (we discussed the motivations for them in [28]). Below, we discuss how those features interact with congruence closure and suggest how the algorithm could be adapted to other settings.

First, our equality is *very heterogeneous*, that is we can form and use equations between terms of different types. This has pros and cons: it can be convenient for the programmer to not worry about types, and the metatheory is simple, but it makes it hard to include type-directed η-rules. However, congruence closure will work just as well with a conventional homogeneous equality.

In fact, in one way a conventionally typed equality would work better, because if would allow a more expressive congruence rule. In first-order logic, a term is either an atom or an application, so there is just a single congruence rule, the one for applications. One might expect that our relation would have one congruence rule for each syntactic form (i.e. for $a = b$ and $(x : A) \to B$ and $\mathsf{rec}\ f\ x.a$ etc). However, we do not do that, because it would lead to problems for terms with variable-binding structure. For those, one would expect the congruence rules to go under binders, e.g.:

$$\frac{\Gamma, x : A \vDash b = b'}{\Gamma \vDash (\lambda x_A.b) = (\lambda x_A.b')}$$

However, adding this rule is equivalent to adding functional extensionality, which is not compatible with our "very heterogeneous" treatment of equality [28]. Instead we adopt the rule TCCCONGRUENCE, which is phrased in terms of substitution. This is rule in particular subsumes the usual congruence rule for application, but it additionally allows changing subterms under binders, as long as the subterms do not mention the bound variables.

Second, we use an *n-ary congruence rule*, while most theories only allow eliminating one equation at a time. For congruence closure to work equality must be a congruence, e.g. given $a = a'$ and $b = b'$ we should be able to conclude $f\ a\ b = f\ a'\ b'$. Our n-ary rule supports this in the most straightforward way possible. An alternative (used in some versions of ETT [13]) would be to use separate n-ary congruence rules for each syntactic form. Systems that only allow rewriting by one equation at a time require some tricks to avoid ill-typed intermediate terms (e.g. [7] Section 8.2.7).

Finally, in our system the elimination of propositional equality is *erased*, so equations like $a_{\triangleright b} = a$ are considered trivially true. This is similar to Extensional Type Theory, but unlike Coq and Agda. Having such equations available is important, because the elaborator inserts casts automatically, without detailed control by the programmer. In Coq that would be problematic, because an inserted cast could prevent two terms from being equal. However, making the conversion erasable is not the only possible approach. For example, in Observational Type Theory [2] the conversions are computationally relevant but the theory includes $a_{\triangleright b} = a$ as an axiom. In that system one can imagine the elaborator would use the axiom to make the elaborated program type-check.

Stronger equational theories The theory of congruence closure is one among a number of related theories. One can strengthen it in various ways by adding more reasoning rules, in order to get a more expressive programming language. However, doing so may endanger type inference, or even the decidability of type checking.

One obvious question is whether we could extend the relation $\Gamma \vDash a = b$ to do both congruence reasoning and β-reduction at the same time. Unfortunately, this extension causes the relation to become undecidable.

This is clearly the case in our language, which directly includes general recursive function definitions. But even if we allowed only terminating functions, the combination of equality assumption and lambdas can be used to encode general recursion. For example, reasoning in a context containing

```
f : Nat → Nat
h : f = (\x. if (even x) then f (n/2) else f (3*n+1))
```

is equivalent to having available a direct recursive definition

```
f x = if (even x) then f (n/2) else f (3*n+1)
```

Another natural generalization is to allow *rewriting by axiom schemes*, i.e. instead of only using ground equations $a = b$ from the context, also instantiate and use quantified formulas like $\forall xyz.a = b$. In general this generalization (the "word problem") is also not decidable, e.g. it is easy to write down an axiom scheme for the equational theory of SKI-combinators. However, there are semi-decision procedures such as *unfailing completion* [5] which form the basis of many automated theorem provers.

Even when preserving decidability one can still extend congruence closure to know about specific axioms schemes, such as for natural numbers with successor and predecessor [23] or lists [22] or injective data constructors [12].

Clearly one could design a programming language around a more ambitious theory than just congruence closure. Many languages, such as Dafny [19] and Dminor [8] call out to an off-the-shelf theorem prover in order to take advantage of all the theories that the prover implements. One reason we focus on a simple theory is that it makes *unification* easier, which seems to offer promising avenues for future work on type inference. Unification modulo congruence closure (rigid E-unification) is NP-complete [17]. This compares favorably with unification modulo β (higher-order unification) which is undecidable. Unification modulo other equational theories (E-unification) must be handled on a theory-by-theory basis, and it is not an operation exposed by most provers.

Simplifying congruence proofs Our CONGTRANS simplification rule is quite natural, and in fact the same rule has been studied before for a different reason. For efficiency, users of congruence closure want to make proofs as small as possible by taking advantage of simplifications like $\mathsf{refl}; p \mapsto p$ or $p^{-1}; p \mapsto \mathsf{refl}$ [15, 31]. However, uses of cong can hide the opportunity for such simplifications. De Moura et al. define the same CONGTRANS rule and give the following example [15]. Given assumptions $h_1 : a = b, h_2 : b = d, h_3 : c = b$, consider the proof term

$$(\mathsf{cong}_f\ (h_1; {h_3}^{-1})); (\mathsf{cong}_f\ (h_3; h_2)) : fa = fd$$

We can get rid of the assumption h_3 by doing the rewrite

$$(\mathsf{cong}_f\ (h_1; {h_3}^{-1})); (\mathsf{cong}_f\ (h_3; h_2)) \mapsto \mathsf{cong}_f\ (h_1; {h_3}^{-1}; h_3; h_2).$$

Dependent programming with congruence closure CoqMT [30] aims to make Coq's definitional equality stronger by including additional equational theories, such as Presburger arithmetic, so that for example the types $\mathsf{Vec}\ 0$ and $\mathsf{Vec}\ (n \times 0)$ can be used interchangeably. The prototype implementation only looks at the types themselves, but the metatheory also considers using assumptions from the context. This is complicated because CoqMT still wants to consider types modulo β-convertibility, and in contexts with inconsistent assumptions like $\mathsf{true} = \mathsf{false}$ one could write nonterminating expressions. Therefore CoqMT imposes restrictions on where an assumption can be used. VeriML makes the definitional equality user-programmable [29], and as an example builds a "stack" combining congruence closure, β-reduction, and potentially other theorem proving.

Neither CoqMT or VeriML prove that their implementation is complete with respect to a declarative specification. For example, the VeriML application rule requires that the applied function has the type $T \to T'$ and then checks that T is definitionally equal to the type of the argument, but there is no attempt to also handle declarative derivations which require definitional equality to create an arrow type.

The Guru language includes a tactic `hypjoin` [24] similar to our `smartjoin` and `unfold`. However, instead of using equations from the context, the programmer has to write an explicit list of equations, and unlike `unfold` it normalizes the given equations.

10. Conclusion

We consider this paper as an application of automatic theorem proving to language design. Of course, in a higher-order logic, we always expect that the programmer will have to supply *some* proofs manually; the question is which ones. Intensional Type Theory recognizes that $\beta\eta$-equivalence in a normalizing language is decidable, so such equality proofs can be handled automatically as part of the definitional equality relation. This paper considers a different decidable equational theory, and proposes a language that is "the dual of ITT": while conventional dependently-typed languages automatically use equalities that follow from β-reductions but do not automatically use assumptions from the context, our language uses assumptions but does not automatically reduce expressions.

We look forward to exploring the ramifications of this design decision more deeply in the context of a full programming language. Our ZOMBIE implementation provides a good baseline, but we would like to add more automation. In particular, the addition of rigid E-unification seems promising. Furthermore, we would like to explore ways in which β-reduction and congruence closure can co-exist—perhaps there is some way to achieve the benefits of each approach in the same context.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 0910500, 1116620, and 1319880. The ZOMBIE implementation was developed with the assistance of the Trellys team. This paper was written with the help of the Ott tool [26]. The authors would also like to thank the anonymous reviewers for their comments.

References

[1] T. Altenkirch. The case of the smart case: How to implement conditional convertibility? Presentation at NII Shonan seminar 007, Japan, Sept. 2011.

[2] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *PLPV '07: Programming Languages meets Program Verification*, pages 57–68. ACM, 2007.

[3] L. Augustsson. Cayenne – a language with dependent types. In *ICFP '98: International Conference on Functional Programming*, pages 239–250. ACM, 1998.

[4] L. Bachmair and A. Tiwari. Abstract congruence closure and specializations. In D. McAllester, editor, *Automated Deduction — CADE-17*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 64–78. Springer-Verlag, jun 2000.

[5] L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion Without Failure. In A. H. Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 1–30. Academic Press, 1989.

[6] B. Barras and B. Bernardo. The Implicit Calculus of Constructions as a Programming Language with Dependent Types. In *11th international conference on Foundations of Software Science and Computational Structures (FOSSACS 2008)*, volume 4962 of *LNCS*, pages 365–379. Springer, 2008.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[8] G. M. Bierman, A. D. Gordon, C. Hritcu, and D. E. Langworthy. Semantic subtyping with an SMT solver. In *ICFP '10: International Conference on Functional Programming*, pages 105–116, 2010.

[9] E. C. Brady. Idris—systems programming meets full dependent types. In *PLPV'11: Programming languages meets program verification*, pages 43–54. ACM, 2011. ISBN 978-1-4503-0487-0.

[10] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed langauge. In *POPL '14: 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

[11] P. Corbineau. *Démonstration automatique en Théorie des Types*. PhD thesis, University Paris 11, September 2005.

[12] P. Corbineau. Deciding equality in the constructor theory. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 78–92. Springer Berlin Heidelberg, 2007.

[13] K. Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Cornell University, 1998.

[14] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] L. de Moura, H. Rueß, and N. Shankar. Justifying equality. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 125(3):69–85, July 2005.

[16] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, Oct. 1980.

[17] J. Gallier, W. Snyder, P. Narendran, and D. Plaisted. Rigid E-unification is NP-complete. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88)*, pages 218–227, 1988.

[18] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. AURA: A programming language for authorization and audit. In *ICFP '08: International Conference on Functional Programming)*, pages 27–38, 2008.

[19] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th international conference on Logic for programming, artificial intelligence, and reasoning*, LPAR'10, pages 348–370. Springer-Verlag, 2010.

[20] C. McBride. First-order unification by structural recursion, 2001.

[21] S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in agda: dependent types for relational program derivation. 19(5):545–579, 2009.

[22] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, Apr. 1980.

[23] R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, Apr. 2007.

[24] A. Petcher and A. Stump. Deciding Joinability Modulo Ground Equations in Operational Type Theory. In S. Lengrand and D. Miller, editors, *Proof Search in Type Theories (PSTT)*, 2009.

[25] B. C. Pierce and D. N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, Jan. 2000.

[26] P. Sewell, F. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: Effective tool support for the working semanticist. *J. Funct. Program.*, 20(1):71–122, 2010.

[27] R. E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, July 1978.

[28] V. Sjöberg, C. Casinghino, K. Y. Ahn, N. Collins, H. D. Eades III, P. Fu, G. Kimmell, T. Sheard, A. Stump, and S. Weirich. Irrelevance, heterogeneous equality, and call-by-value dependent type systems. In J. Chapman and P. B. Levy, editors, *MSFP '12*, volume 76 of *EPTCS*, pages 112–162. Open Publishing Association, 2012.

[29] A. Stampoulis and Z. Shao. Static and user-extensible proof checking. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 273–284, New York, NY, USA, 2012. ACM.

[30] P.-Y. Strub. Coq modulo theory. In *CSL*, pages 529–543, 2010.

[31] A. Stump and L.-Y. Tan. The algebra of equality proofs. In *16th International Conference on Rewriting Techniques and Applications (RTA'05)*, pages 469–483. Springer, 2005.

[32] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. Secure Distributed Programming with Value-dependent Types. In *ICFP '11: International Conference on Functional Programming*, pages 285–296. ACM, 2011.

A. Full specification

A.1 Grammars

$env,\ \Gamma$	$::=$		typing environment
	$\mid$	$\cdot$	empty
	$\mid$	$\Gamma,\ decl$	
$decl$	$::=$		typing env declaration
	$\mid$	$x : A$	variable
$exp,\ a,\ b,\ c,\ A,\ B,\ C$	$::=$		annotated expressions
	$\mid$	Type	type
	$\mid$	x	variable
	$\mid$	$\mathsf{rec}\ f_A\ x.a$	recursive definition
	$\mid$	$\mathsf{rec}\ f_A\ \bullet_x.a$	irrelevant recursive definition
	$\mid$	$a\ b$	application
	$\mid$	$a\ \bullet_b$	irrelevant application
	$\mid$	$(x:A) \to B$	function type
	$\mid$	$\bullet(x:A) \to B$	irrelevant function type
	$\mid$	$a = b$	equality proposition
	$\mid$	join_Σ	equality proof
	$\mid$	$a_{\triangleright b}$	type conversion
Σ	$::=$		equality strategies
	$\mid$	$\leadsto_{\mathsf{cbv}}\ i\, j : a = b$	
	$\mid$	$\leadsto_{\mathsf{p}}\ i\, j : a = b$	
	$\mid$	$\bullet$	
	$\mid$	$\mathsf{injdom}\ a$	
	$\mid$	$\mathsf{injrng}\ a\ b$	
	$\mid$	$\mathsf{injeq}\ i\ a$	
	$\mid$	$CTX : B$	
$ctx,\ CTX$	$::=$		subst context
	$\mid$	$\{\sim b_1/x_1\} \ldots \{\sim b_i/x_i\} A$	
$val,\ V,\ v$	$::=$		values
	$\mid$	x	
	$\mid$	Type	
	$\mid$	$(x:A) \to B$	
	$\mid$	$\bullet(x:A) \to B$	
	$\mid$	$a = b$	
	$\mid$	$v_{\triangleright v'}$	
	$\mid$	join_Σ	
	$\mid$	$\mathsf{rec}\ f_A\ x.a$	
	$\mid$	$\mathsf{rec}\ f_A\ \bullet_x.a$	

A.2 Core language specification

The following rules define the full type system for the core language of the paper. It includes two sets of rules which were omitted from the body of the paper for space. First, in addition to join proofs $\mathsf{join}_{\leadsto_{\mathsf{cbv}} i\, j : a=b}$, which prove goals using a parallel reduction relationship, we also allow $\mathsf{join}_{\leadsto_{\mathsf{cbv}} i\, j : a=b}$, which uses plain CBV-evaluation (without reducing under binders). The relation $\leadsto_{\mathsf{p}}$ will prove more goals, but when it works we expect $\leadsto_{\mathsf{cbv}}$ to be more efficient. (This is analogous to how Coq provides both `lazy` and `cbv` evaluation tactics).

Second, we include typing rules for computationally irrelevant function (TIPI, TIREC, and TIDAPP). These are similar to the rules for ordinary functions, except that TIREC has an additional restriction that the argument x must not appear in a computationally relevant position.

$\boxed{a \leadsto_{\mathsf{cbv}} b}$

$$\frac{}{(\mathsf{rec}\ f\ x.a)\ v \leadsto_{\mathsf{cbv}} \{v/x\}\,\{\mathsf{rec}\ f\ x.a/f\}\ a}\ \textsc{SCappbeta}$$

$$\frac{}{(\mathsf{rec}\ f\ \bullet.a)\ \bullet \leadsto_{\mathsf{cbv}} \{\mathsf{rec}\ f\ \bullet.a/f\}\ a}\ \textsc{SCiappbeta}$$

$$\frac{a \rightsquigarrow_{\mathsf{cbv}} a'}{a\ b \rightsquigarrow_{\mathsf{cbv}} a'\ b}\textsc{SCctx1}$$

$$\frac{a \rightsquigarrow_{\mathsf{cbv}} a'}{v\ a \rightsquigarrow_{\mathsf{cbv}} v\ a'}\textsc{SCctx2}$$

$$\frac{a \rightsquigarrow_{\mathsf{cbv}} a'}{a\ \bullet \rightsquigarrow_{\mathsf{cbv}} a'\ \bullet}\textsc{SCctx3}$$

$\boxed{a \rightsquigarrow_{\mathsf{p}} b}$

$$\frac{}{a \rightsquigarrow_{\mathsf{p}} a}\textsc{SPrefl}$$

$$\frac{a \rightsquigarrow_{\mathsf{p}} a'}{\mathsf{rec}\ f\ x.a \rightsquigarrow_{\mathsf{p}} \mathsf{rec}\ f\ x.a'}\textsc{SPrec}$$

$$\frac{\begin{array}{c}A \rightsquigarrow_{\mathsf{p}} A' \\ B \rightsquigarrow_{\mathsf{p}} B'\end{array}}{(x\!:\!A) \to\ B \rightsquigarrow_{\mathsf{p}} (x\!:\!A') \to\ B'}\textsc{SPpi}$$

$$\frac{\begin{array}{c}A \rightsquigarrow_{\mathsf{p}} A' \\ B \rightsquigarrow_{\mathsf{p}} B'\end{array}}{\bullet(x\!:\!A) \to\ B \rightsquigarrow_{\mathsf{p}} \bullet(x\!:\!A') \to\ B'}\textsc{SPipi}$$

$$\frac{\begin{array}{c}a \rightsquigarrow_{\mathsf{p}} a' \\ b \rightsquigarrow_{\mathsf{p}} b'\end{array}}{a = b \rightsquigarrow_{\mathsf{p}} a' = b'}\textsc{SPeq}$$

$$\frac{\begin{array}{c}a \rightsquigarrow_{\mathsf{p}} a' \\ b \rightsquigarrow_{\mathsf{p}} b'\end{array}}{a\ b \rightsquigarrow_{\mathsf{p}} a'\ b'}\textsc{SPapp}$$

$$\frac{\begin{array}{c}a \rightsquigarrow_{\mathsf{p}} a' \\ v \rightsquigarrow_{\mathsf{p}} v'\end{array}}{(\mathsf{rec}\ f\ x.a)\ v \rightsquigarrow_{\mathsf{p}} (\{v'/x\}\,\{\mathsf{rec}\ f\ x.a'/f\}\ a')}\textsc{SPappbeta}$$

$$\frac{a \rightsquigarrow_{\mathsf{p}} a'}{(\mathsf{rec}\ f\ \bullet.a)\ \bullet \rightsquigarrow_{\mathsf{p}} (\{\mathsf{rec}\ f\ \bullet.a'/f\}\ a')}\textsc{SPiappbeta}$$

$\boxed{\Gamma \vdash a : A}$ Annotated core language typing rules

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{Type} : \mathsf{Type}}\textsc{Ttype}$$

$$\frac{x : A \in \Gamma \quad \vdash \Gamma}{\Gamma \vdash x : A}\textsc{Tvar}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \mathsf{Type} \\ \Gamma, x : A \vdash B : \mathsf{Type}\end{array}}{\Gamma \vdash (x\!:\!A) \to\ B : \mathsf{Type}}\textsc{Tpi}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \mathsf{Type} \\ \Gamma, x : A \vdash B : \mathsf{Type}\end{array}}{\Gamma \vdash \bullet(x\!:\!A) \to\ B : \mathsf{Type}}\textsc{Tipi}$$

$$\frac{\begin{array}{c}\Gamma \vdash (x\!:\!A_1) \to\ A_2 : \mathsf{Type} \\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash a : A_2\end{array}}{\Gamma \vdash \mathsf{rec}\ f_{(x A_1) \to\ A_2}\ x.a : (x\!:\!A_1) \to\ A_2}\textsc{Trec}$$

$$\frac{\begin{array}{l}\Gamma \vdash \bullet(x\!:\!A_1) \to\ A_2 : \mathsf{Type} \\ \Gamma, f : \bullet(x\!:\!A_1) \to\ A_2, x : A_1 \vdash a : A_2 \\ x\ \notin \mathsf{FV}\,(|a|)\end{array}}{\Gamma \vdash \mathsf{rec}\ f_{(x A_1) \to\ A_2}\ \bullet_x.a : (x\!:\!A_1) \to\ A_2}\textsc{Tirec}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : A \to B \\ \Gamma \vdash b : A\end{array}}{\Gamma \vdash a\ b : B}\textsc{Tapp}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : (x\!:\!A) \to\ B \\ \Gamma \vdash v : A\end{array}}{\Gamma \vdash a\ v : \{v/x\}\,B}\textsc{Tdapp}$$

$$\frac{\begin{array}{l}\Gamma \vdash a : \bullet(x\!:\!A) \to B\\ \Gamma \vdash v : A\end{array}}{\Gamma \vdash a \bullet_v : \{v/x\}\, B}\ \textsc{TIdApp}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash a = b : \mathsf{Type}}\ \textsc{TEq}$$

$$\frac{|a_1| \leadsto^{i}_{\mathsf{cbv}} b \quad |a_2| \leadsto^{j}_{\mathsf{cbv}} b \quad \Gamma \vdash a_1 = a_2 : \mathsf{Type}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}}\, i\ j : a_1 = a_2} : a_1 = a_2}\ \textsc{TJoinC}$$

$$\frac{|a_1| \leadsto^{i}_{\mathsf{p}} b \quad |a_2| \leadsto^{j}_{\mathsf{p}} b \quad \Gamma \vdash a_1 = a_2 : \mathsf{Type}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{p}}\, i\ j : a_1 = a_2} : a_1 = a_2}\ \textsc{TJoinP}$$

$$\frac{\Gamma \vdash v : ((x\!:\!A_1) \to B_1) = ((x\!:\!A_2) \to B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injdom}\ v} : A_1 = A_2}\ \textsc{TJInjDom}$$

$$\frac{\Gamma \vdash v_1 : ((x\!:\!A) \to B_1) = ((x\!:\!A) \to B_2) \quad \Gamma \vdash v_2 : A}{\Gamma \vdash \mathsf{join}_{\mathsf{injrng}\ v_1\ v_2} : \{v_2/x\}\, B_1 = \{v_2/x\}\, B_2}\ \textsc{TJInjRng}$$

$$\frac{\Gamma \vdash v : (\bullet(x\!:\!A_1) \to B_1) = (\bullet(x\!:\!A_2) \to B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injdom}\ v} : A_1 = A_2}\ \textsc{TJIInjDom}$$

$$\frac{\Gamma \vdash v_1 : (\bullet(x\!:\!A) \to B_1) = (\bullet(x\!:\!A) \to B_2) \quad \Gamma \vdash v_2 : A}{\Gamma \vdash \mathsf{join}_{\mathsf{injrng}\ v_1\ v_2} : \{v_2/x\}\, B_1 = \{v_2/x\}\, B_2}\ \textsc{TJIInjRng}$$

$$\frac{\Gamma \vdash v : (A_1 = A_2) = (B_1 = B_2)}{\Gamma \vdash \mathsf{join}_{\mathsf{injeq}\ i\ v} : A_i = B_i}\ \textsc{TJInjEq}$$

$$\frac{\begin{array}{l}\Gamma \vdash B : \mathsf{Type} \quad \forall k.\ \Gamma \vdash v_k : a_k = b_k\\ |B| = |(\{a_1/x_1\}\ldots\{a_j/x_j\}\, c = \{b_1/x_1\}\ldots\{b_j/x_j\}\, c)|\end{array}}{\Gamma \vdash \mathsf{join}_{\{\sim v_1/x_1\}\ldots\{\sim v_j/x_j\}\, c : B} : B}\ \textsc{TJSubst}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash v : A = B \quad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash a_{\triangleright v} : B}\ \textsc{TCast}$$

$\boxed{\vdash \Gamma}$ G is a well-formed environment

$$\frac{}{\vdash \cdot}\ \textsc{EnvWfEmpty}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash A : \mathsf{Type}}{\vdash \Gamma, x : A}\ \textsc{EnvWfVar}$$

A.3 Congruence closure and injrng for core language

$\boxed{\Gamma \vDash a = b}$ typed congruence closure (up to erasure)

$$\frac{\Gamma \vdash a : A}{\Gamma \vDash a = a}\ \textsc{TCCRefl}$$

$$\frac{\Gamma \vDash a = b}{\Gamma \vDash b = a}\ \textsc{TCCSym}$$

$$\frac{\Gamma \vDash a = b \quad \Gamma \vDash b = c}{\Gamma \vDash a = c}\ \textsc{TCCTrans}$$

$$\frac{x : A \in \Gamma \quad \Gamma \vDash A = (a = b)}{\Gamma \vDash a = b}\ \textsc{TCCAssumption}$$

$$\frac{\begin{array}{l}\Gamma \vdash A = B : \mathsf{Type} \quad \forall k.\ \Gamma \vDash a_k = b_k\\ |A = B| = |\{a_1/x_1\}\ldots\{a_j/x_j\}\, c = \{b_1/x_1\}\ldots\{b_j/x_j\}\, c|\end{array}}{\Gamma \vDash A = B}\ \textsc{TCCCongruence}$$

$$\frac{\Gamma \vDash ((x\!:\!A_1) \to B_1) = ((x\!:\!A_2) \to B_2)}{\Gamma \vDash A_1 = A_2}\ \textsc{TCCInjDom}$$

$$\frac{\Gamma \vDash (A_1 \to B_1) = (A_2 \to B_2)}{\Gamma \vDash B_1 = B_2}\ \textsc{TCCInjRng}$$

$$\frac{\Gamma \vDash (\bullet(x\!:\!A_1) \to\ B_1) = (\bullet(x\!:\!A_2) \to\ B_2)}{\Gamma \vDash A_1 = A_2}\ \textsc{TCCiinjdom}$$

$$\frac{\Gamma \vDash (\bullet A_1 \to\ B_1) = (\bullet A_2 \to\ B_2)}{\Gamma \vDash B_1 = B_2}\ \textsc{TCCiinjrng}$$

$$\frac{\Gamma \vDash (a_1 = a_2) = (b_1 = b_2)}{\Gamma \vDash a_k = b_k}\ \textsc{TCCinjeq}$$

$$\frac{|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vDash a = b}\ \textsc{TCCerasure}$$

$\boxed{\Gamma \vDash \ \mathsf{injrng}\ A\ \mathsf{for}\ v}$ injectivity side condition

$$\frac{\begin{array}{l}\Gamma \vdash v : A \quad \Gamma \vdash (x\!:\!A) \to\ B : \mathsf{Type} \\ \forall A'\ B'.((\Gamma \vDash ((x\!:\!A) \to\ B) = ((x\!:\!A') \to\ B')) \text{ implies } (\Gamma \vDash \{v/x\}\,B = \{v_{\triangleright v_0}/x\}\,B' \text{ where } \Gamma \vdash v_0 : A = A'))\end{array}}{\Gamma \vDash \ \mathsf{injrng}\ (x\!:\!A) \to\ B\ \mathsf{for}\ v}\ \textsc{IRpi}$$

$$\frac{\begin{array}{l}\Gamma \vdash v : A \quad \Gamma \vdash \bullet(x\!:\!A) \to\ B : \mathsf{Type} \\ \forall A'\ B'.((\Gamma \vDash (\bullet(x\!:\!A) \to\ B) = (\bullet(x\!:\!A') \to\ B')) \text{ implies } (\Gamma \vDash \{v/x\}\,B = \{v_{\triangleright v_0}/x\}\,B' \text{ where } \Gamma \vdash v_0 : A = A'))\end{array}}{\Gamma \vDash \ \mathsf{injrng}\ \bullet(x\!:\!A) \to\ B\ \mathsf{for}\ v}\ \textsc{IRipi}$$

A.4 Bidirectional system

$\boxed{\Gamma \vdash a \Rightarrow A}$ Inference mode

$$\frac{\vdash \Gamma \Leftarrow}{\Gamma \vdash \mathsf{Type} \Rightarrow \mathsf{Type}}\ \textsc{Itype}$$

$$\frac{\vdash \Gamma \Leftarrow \quad x : A \in \Gamma \quad \Gamma \vdash A \Leftarrow \mathsf{Type}}{\Gamma \vdash x \Rightarrow A}\ \textsc{Ivar}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma, x : A \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash (x\!:\!A) \to\ B \Rightarrow \mathsf{Type}}\ \textsc{Ipi}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma, x : A \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash \bullet(x\!:\!A) \to\ B \Rightarrow \mathsf{Type}}\ \textsc{Iipi}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow (x\!:\!A) \to\ B \quad \Gamma \vdash v \Leftarrow A \\ \Gamma \vDash^{\exists}\ \mathsf{injrng}\ (x\!:\!A) \to\ B\ \mathsf{for}\ v \\ \Gamma \vdash \{v_A/x\}\,B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a\ v \Rightarrow \{v_A/x\}\,B}\ \textsc{Idapp}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow \bullet(x\!:\!A) \to\ B \\ \Gamma \vdash v \Leftarrow A \\ \Gamma \vDash^{\exists}\ \mathsf{injrng}\ \bullet(x\!:\!A) \to\ B\ \mathsf{for}\ v \\ \Gamma \vdash \{v_A/x\}\,B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a\ \bullet_v \Rightarrow \{v_A/x\}\,B}\ \textsc{Iidapp}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow A \to B \quad \Gamma \vdash b \Leftarrow A \\ \Gamma \vdash B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash a\ b \Rightarrow B}\ \textsc{Iapp}$$

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \vdash b \Rightarrow B}{\Gamma \vdash a = b \Rightarrow \mathsf{Type}}\ \textsc{Ieq}$$

$$\frac{\begin{array}{l}\Gamma \vdash a_1 = a_2 \Leftarrow \mathsf{Type} \\ |a_1| \rightsquigarrow^{i}_{\mathsf{cbv}}\ b \quad |a_2| \rightsquigarrow^{j}_{\mathsf{cbv}}\ b\end{array}}{\Gamma \vdash \mathsf{join}_{\rightsquigarrow_{\mathsf{cbv}}\, i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\ \textsc{Ijoinc}$$

$$\frac{\begin{array}{l}\Gamma \vdash a_1 = a_2 \Leftarrow \mathsf{Type} \\ |a_1| \rightsquigarrow^{i}_{\mathsf{p}}\ b \quad |a_2| \rightsquigarrow^{j}_{\mathsf{p}}\ b\end{array}}{\Gamma \vdash \mathsf{join}_{\rightsquigarrow_{\mathsf{p}}\, i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\ \textsc{Ijoinp}$$

$$\frac{\Gamma \vdash A \Leftarrow \mathsf{Type} \quad \Gamma \vdash a \Leftarrow A}{\Gamma \vdash a_A \Rightarrow A}\ \textsc{Iannot}$$

$$\frac{\Gamma \vdash a \Rightarrow A \quad \Gamma \vDash^{\exists} A = B \quad \Gamma \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash a \Rightarrow B}\ \textsc{Icast}$$

$\boxed{\Gamma \vdash a \Leftarrow A}$ Checking mode

$$\dfrac{\begin{array}{l}\Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vdash a \Leftarrow A_2 \\ \Gamma, f : (x\!:\!A_1) \to\ A_2, x : A_1 \vDash^{\exists} \ \mathsf{injrng}\,(x\!:\!A_1) \to\ A_2 \ \mathsf{for}\ x \\ \Gamma, f : (x\!:\!A_1) \to\ A_2 \vdash (x\!:\!A_1) \to\ A_2 \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash \mathsf{rec}\ f\ x.a \Leftarrow (x\!:\!A_1) \to\ A_2}\ \textsc{CRec}$$

$$\dfrac{\begin{array}{l}\Gamma, f : \bullet(x\!:\!A_1) \to\ A_2, x : A_1 \vdash a \Leftarrow A_2 \\ x \notin \mathsf{FV}\,(|a|) \\ \Gamma, f : \bullet(x\!:\!A_1) \to\ A_2, x : A_1 \vDash^{\exists} \ \mathsf{injrng}\ \bullet(x\!:\!A_1) \to\ A_2 \ \mathsf{for}\ x \\ \Gamma, f : \bullet(x\!:\!A_1) \to\ A_2 \vdash \bullet(x\!:\!A_1) \to\ A_2 \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash \mathsf{rec}\ f\ \bullet.a \Leftarrow \bullet(x\!:\!A_1) \to\ A_2}\ \textsc{CIrec}$$

$$\dfrac{\Gamma \vDash^{\exists} a = b}{\Gamma \vdash \mathsf{join} \Leftarrow a = b}\ \textsc{CRefl}$$

$$\dfrac{\Gamma \vdash a \Rightarrow A}{\Gamma \vdash a \Leftarrow A}\ \textsc{CInf}$$

$$\dfrac{\Gamma \vdash a \Leftarrow A \quad \Gamma \vDash^{\exists} A = B \quad \Gamma \vdash B \Leftarrow \mathsf{Type}}{\Gamma \vdash a \Leftarrow B}\ \textsc{CCast}$$

$\boxed{\vdash \Gamma \Leftarrow}$ G is a well-formed environment

$$\dfrac{}{\vdash \cdot \Leftarrow}\ \textsc{GEmpty}$$

$$\dfrac{\begin{array}{l}\vdash \Gamma \quad x \notin \mathsf{dom}\,(\Gamma) \\ \Gamma \vdash A \Leftarrow \mathsf{Type}\end{array}}{\vdash \Gamma, x : A \Leftarrow}\ \textsc{GVar}$$

A.5 Elaboration

$\boxed{\mapsto \Gamma \rightsquigarrow \Gamma'}$ G is a well-formed environment (elaborating version)

$$\dfrac{}{\mapsto \cdot \rightsquigarrow \cdot}\ \textsc{EGEmpty}$$

$$\dfrac{\begin{array}{l}\mapsto \Gamma \rightsquigarrow \Gamma' \\ x \notin \mathsf{dom}\,(\Gamma) \\ \Gamma' \mapsto A \Leftarrow \mathsf{Type} \rightsquigarrow A'\end{array}}{\mapsto \Gamma, x : A \rightsquigarrow \Gamma', x : A'}\ \textsc{EGVar}$$

$\boxed{\Gamma \mapsto a \Rightarrow a' : A}$ Inference mode, with elaboration

$$\dfrac{}{\Gamma \mapsto \mathsf{Type} \Rightarrow \mathsf{Type} : \mathsf{Type}}\ \textsc{EIType}$$

$$\dfrac{x : A \in \Gamma \quad \Gamma \mapsto A \Leftarrow \mathsf{Type} \rightsquigarrow A_0}{\Gamma \mapsto x \Rightarrow x : A}\ \textsc{EIVar}$$

$$\dfrac{\begin{array}{l}\Gamma \mapsto A \Leftarrow \mathsf{Type} \rightsquigarrow A' \\ \Gamma, x : A' \mapsto B \Leftarrow \mathsf{Type} \rightsquigarrow B'\end{array}}{\Gamma \mapsto (x\!:\!A) \to\ B \Rightarrow (x\!:\!A') \to\ B' : \mathsf{Type}}\ \textsc{EIPi}$$

$$\dfrac{\begin{array}{l}\Gamma \mapsto A \Leftarrow \mathsf{Type} \rightsquigarrow A' \\ \Gamma, x : A' \mapsto B \Leftarrow \mathsf{Type} \rightsquigarrow B'\end{array}}{\Gamma \mapsto \bullet(x\!:\!A) \to\ B \Rightarrow \bullet(x\!:\!A') \to\ B' : \mathsf{Type}}\ \textsc{EIIpi}$$

$$\dfrac{\begin{array}{l}\Gamma \mapsto a \Rightarrow a' : A_1 \\ \Gamma \mapsto A_1 =^{?} A \to B \rightsquigarrow v_1 \\ \Gamma \mapsto b \Leftarrow A \rightsquigarrow b'\end{array}}{\Gamma \mapsto a\ b \Rightarrow a'_{\rhd v_1}\ b' : B}\ \textsc{EIApp}$$

$$\dfrac{\begin{array}{l}\Gamma \mapsto a \Rightarrow a' : A_1 \quad \Gamma \mapsto A_1 =^{?} (x : A) \to B \rightsquigarrow v_1 \\ \Gamma \mapsto v \Leftarrow A \rightsquigarrow v' \quad \Gamma \vDash \ \mathsf{injrng}\,(x\!:\!A) \to\ B \ \mathsf{for}\ v'\end{array}}{\Gamma \mapsto a\ v \Rightarrow a'_{\rhd v_1}\ v' : \{v'/x\}\,B}\ \textsc{EIDapp}$$

$$\frac{\begin{array}{l}\Gamma \vdash a \Rightarrow a' : A_1 \\ \Gamma \vdash A_1 =^? [x : A] \to B \leadsto v_1 \\ \Gamma \vdash v \Leftarrow A \leadsto v' \\ \Gamma \vDash \mathsf{injrng}\ \bullet (x : A) \to\ B\ \mathsf{for}\ v'\end{array}}{\Gamma \vdash a\ \bullet_v \Rightarrow a'_{\triangleright v_1}\ \bullet_{v'} : \{v'/x\}\, B}\ \text{EIIDAPP}$$

$$\frac{\Gamma \vdash a \Rightarrow a' : A \quad \Gamma \vdash b \Rightarrow b' : B}{\Gamma \vdash a = b \Rightarrow a' = b' : \mathsf{Type}}\ \text{EIEQ}$$

$$\frac{\begin{array}{l}|a| \leadsto^i_{\mathsf{cbv}} c \quad |b| \leadsto^j_{\mathsf{cbv}} c \\ \Gamma \vdash a = b \Leftarrow \mathsf{Type} \leadsto a' = b'\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}}\, i\, j : a = b} \Rightarrow \mathsf{join}_{\leadsto_{\mathsf{cbv}}\, i\, j : a' = b'} : a' = b'}\ \text{EIJOINC}$$

$$\frac{\begin{array}{l}|a| \leadsto^i_{\mathsf{p}} c \quad |b| \leadsto^j_{\mathsf{p}} c \\ \Gamma \vdash a = b \Leftarrow \mathsf{Type} \leadsto a' = b'\end{array}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{p}}\, i\, j : a = b} \Rightarrow \mathsf{join}_{\leadsto_{\mathsf{p}}\, i\, j : a' = b'} : a' = b'}\ \text{EIJOINP}$$

$$\frac{\begin{array}{l}\Gamma \vdash A \Leftarrow \mathsf{Type} \leadsto A' \\ \Gamma \vdash a \Leftarrow A' \leadsto a'\end{array}}{\Gamma \vdash a_A \Rightarrow a' : A'}\ \text{EIANNOT}$$

$\boxed{\Gamma \vdash a \Leftarrow A \leadsto a'}$ Checking mode, with elaboration

$$\frac{\begin{array}{l}\Gamma \vdash A =^? (x : A_1) \to A_2 \leadsto v_1 \\ \Gamma, f : (x : A_1) \to\ A_2, x : A_1 \vdash a \Leftarrow A_2 \leadsto a' \\ \Gamma, f : (x : A_1) \to\ A_2, x : A_1 \vDash \mathsf{injrng}\, (x : A_1) \to\ A_2\ \mathsf{for}\ x \\ \Gamma, f : (x : A_1) \to\ A_2 \vdash (x : A_1) \to\ A_2 \Leftarrow \mathsf{Type} \leadsto A_0\end{array}}{\Gamma \vdash \mathsf{rec}\ f\ x.a \Leftarrow A \leadsto (\mathsf{rec}\ f_{(xA_1) \to\ A_2}\ x.a')_{\triangleright \mathsf{symm}\ v_1}}\ \text{ECREC}$$

$$\frac{\begin{array}{l}\Gamma \vdash A =^? [x : A_1] \to A_2 \leadsto v_1 \\ \Gamma, f : \bullet(x : A_1) \to\ A_2, x : A_1 \vdash a \Leftarrow A_2 \leadsto a' \\ \Gamma, f : \bullet(x : A_1) \to\ A_2, x : A_1 \vDash \mathsf{injrng}\ \bullet (x : A_1) \to\ A_2\ \mathsf{for}\ x \\ x\ \notin \mathsf{FV}\, (|a'|) \\ \Gamma, f : \bullet(x : A_1) \to\ A_2 \vdash \bullet(x : A_1) \to\ A_2 \Leftarrow \mathsf{Type} \leadsto A_0\end{array}}{\Gamma \vdash \mathsf{rec}\ f\ \bullet.a \Leftarrow A \leadsto (\mathsf{rec}\ f_{\bullet(xA_1) \to\ A_2}\ \bullet_x.a')_{\triangleright \mathsf{symm}\ v_1}}\ \text{ECIREC}$$

$$\frac{\Gamma \vdash A =^? (a = b) \leadsto v_1 \quad \Gamma \vdash a \stackrel{?}{=} b \leadsto v}{\Gamma \vdash \mathsf{join} \Leftarrow A \leadsto v_{\triangleright \mathsf{symm}\ v_1}}\ \text{ECREFL}$$

$$\frac{\Gamma \vdash a \Rightarrow a' : A \quad \Gamma \vdash A \stackrel{?}{=} B \leadsto v_1}{\Gamma \vdash a \Leftarrow B \leadsto a'_{\triangleright v_1}}\ \text{ECINF}$$

B. Assumptions

B.1 Assumptions about the annotated core language

The following properties of the core language were proved in our prior work [28], so in this paper we assume them without proof.

***Assumption* 13** (Weakening for annotated language)**.** If $\Gamma \vdash a : A$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash a : A$.

***Assumption* 14** (Strengthening for annotated language)**.** If $\Gamma, \Gamma' \vdash b : B$ and $\mathsf{FV}\,(b) \subseteq \mathsf{dom}\,(\Gamma)$, then $\Gamma \vdash b : B$.

***Assumption* 15** (Inversion for type well-formedness)**.** 1. If $\Gamma \vdash a = b : C$, then $\Gamma \vdash a = b : \mathsf{Type}$ and there exists A and B such that $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$.

2. If $\Gamma \vdash (x : A) \to B : C$, then $\Gamma \vdash (x : A) \to B : \mathsf{Type}$ and $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma, x : A \vdash B : \mathsf{Type}$.

3. If $\Gamma \vdash \bullet(x : A) \to B : C$, then $\Gamma \vdash \bullet(x : A) \to B : \mathsf{Type}$ and $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma, x : A \vdash B : \mathsf{Type}$.

***Assumption* 16** (Substitution for fully-annotated language)**.** If $\Gamma, x : A \vdash b : B$ and $\Gamma \vdash v : A$, then $\Gamma \vdash \{v/x\}\, b : \{v/x\}\, B$.

***Assumption* 17** (Regularity for fully-annotated language)**.**

If $\vdash \Gamma$ and $x : A \in \Gamma$, then $\Gamma \vdash A : \mathsf{Type}$.

If $\Gamma \vdash a : A$ then $\vdash \Gamma$ and $\Gamma \vdash A : \mathsf{Type}$

B.2 Algorithmic congruence closure relations

Next we specify what assumptions we make about the congruence closure algorithm. Calls to it are represented as judgements:

- $\Gamma \Vdash A =^? (x : A') \to B' \rightsquigarrow v$
- $\Gamma \Vdash A =^? [x : A'] \to B' \rightsquigarrow v$
- $\Gamma \Vdash A =^? (A' = B') \rightsquigarrow v$
- $\Gamma \Vdash A \stackrel{?}{=} B \rightsquigarrow v$

Here, A and B are inputs, while A', B' and v are outputs. For example, $\Gamma \Vdash A =^? (x : A') \to B' \rightsquigarrow v$ means "find A' and B' such that $\Gamma \vDash A = (x : A') \to B'$, and a v such that $\Gamma \vdash v : A = ((x : A') \to B')$". Note that the judgement $\Gamma \Vdash A =^? A' \to B' \rightsquigarrow v$ is syntactic sugar for the dependently-typed version, $\Gamma \Vdash A =^? (x : A') \to B' \rightsquigarrow v$.

By using the congruence closure algorithm presented in Section 7 these relations can be straightforwardly implemented: one constructs the congruence closure of all equations in the context, and then checks whether the equivalence class of A contains any members with the right form (and return the first one found if there are several). Note that this algorithm will give the same answer for two inputs which are in the same equivalence class (but with a different proof v). We formalize that observation as the following assumption.

***Assumption* 18.** (Respects CC) If $\Gamma \vDash A = B$

- $\Gamma \Vdash B \stackrel{?}{=} C \rightsquigarrow v_1$ then $\Gamma \Vdash A \stackrel{?}{=} C \rightsquigarrow v_2$.
- $\Gamma \Vdash B =^? (x : C_1) \to C_2 \rightsquigarrow v_1$ then $\Gamma \Vdash A =^? (x : C_1) \to C_2 \rightsquigarrow v_2$.
- other forms of types

Furthermore, the algorithm can also generate terms in the core language that prove that the required equation holds. We write this as $\Gamma \Vdash A =^? (x : B_1) \to B_2 \rightsquigarrow v$, etc. It is also convenient to specify that the inferred proof always erases to just join. That is, we assume the following interface.

***Assumption* 19.** (CC soundness for function types) If $\Gamma \Vdash A =^? (x : B_1) \to B_2 \rightsquigarrow v$, then $\Gamma \vdash v : A = ((x : B_1) \to B_2)$ and $|v| = \mathsf{join}$ and $\Gamma \vDash A = ((x : B_1) \to B_2)$.

Similar assumptions are required for the other versions of the relation.

Finally we use the elaborating relation $\Gamma \Vdash A \stackrel{?}{=} B \rightsquigarrow v$, which decides whether $\Gamma \vDash A = B$ (both A and B are inputs), and if so produces a core proof term v for the equation.

***Assumption* 20.** (CC soundness) If $\Gamma \Vdash A \stackrel{?}{=} B \rightsquigarrow v$, then $\Gamma \vdash v : A = B$ and $|v| = \mathsf{join}$ and $\Gamma \vDash A = B$.

***Assumption* 21.** (CC completeness) If $\Gamma \vDash A = (x : B_1) \to B_2$ then there exists a $(x : B_1') \to B_2'$ and v such that $\Gamma \Vdash A =^? (x : B_1) \to B_2 \rightsquigarrow v$ succeeds.

Similar assumptions are required for the other versions of the relation.

$$\frac{|a| = |b|}{\Gamma \vdash a = b}\textsc{CCRefl} \qquad \frac{\Gamma \vdash a = b}{\Gamma \vdash b = a}\textsc{CCSym} \qquad \frac{\Gamma \vdash a = b \quad \Gamma \vdash b = c}{\Gamma \vdash a = c}\textsc{CCTrans} \qquad \frac{x : A \in \Gamma \quad \Gamma \vdash A = (a = b)}{\Gamma \vdash a = b}\textsc{CCAssumption}$$

$$\frac{\Gamma \vdash a = b}{\Gamma \vdash \{a/x\}\, c = \{b/x\}\, c}\textsc{CCCongruence} \qquad \frac{\Gamma \vdash (A_1 \to B_1) = (A_2 \to B_2)}{\Gamma \vdash A_1 = B_1}\textsc{CCInjDom} \qquad \frac{\Gamma \vdash (A_1 \to B_1) = (A_2 \to B_2)}{\Gamma \vdash A_2 = B_2}\textsc{CCInjRng}$$

$$\frac{\Gamma \vdash (\bullet A_1 \to B_1) = (\bullet A_2 \to B_2)}{\Gamma \vdash A_1 = B_1}\textsc{CCIInjDom} \qquad \frac{\Gamma \vdash (\bullet A_1 \to B_1) = (\bullet A_2 \to B_2)}{\Gamma \vdash A_2 = B_2}\textsc{CCIInjRng} \qquad \frac{\Gamma \vdash (a_1 = a_2) = (b_1 = b_2)}{\Gamma \vdash a_k = b_k}\textsc{CCInjEq}$$

Figure 14. Untyped congruence closure

$$\frac{}{\Gamma \vdash^{\mathsf{L}} \mathsf{refl} : a = a}\textsc{CCPRefl} \qquad \frac{\Gamma \vdash^{\mathsf{L}} p : a = b}{\Gamma \vdash^{\mathsf{L}} p^{-1} : b = a}\textsc{CCPSym} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} p : a = b\\ \Gamma \vdash^{\mathsf{L}} q : b = c\end{array}}{\Gamma \vdash^{\mathsf{L}} p; q : a = c}\textsc{CCPTrans}$$

$$\frac{\begin{array}{l}x : A \in \Gamma\\ \Gamma \vdash^{\mathsf{L}} p : A = ((- = -)\ a\ b)\end{array}}{\Gamma \vdash^{\mathsf{L}} x_{\triangleright p} : a = b}\textsc{CCPAssumption} \qquad \frac{\forall k.\ \Gamma \vdash^{\mathsf{L}} p_k : a_k = b_k}{\Gamma \vdash^{\mathsf{L}} \mathsf{cong}\, F\, \overline{p_k}^{\,k} : F\,\overline{a_i} = F\,\overline{b_i}}\textsc{CCPCong} \qquad \frac{\begin{array}{l}\Gamma \vdash^{\mathsf{L}} p : F\,\overline{a_i} = F\,\overline{b_i}\\ F\ \mathsf{injective}\end{array}}{\Gamma \vdash^{\mathsf{L}} \mathsf{inj}_i\ p : a_i = b_i}\textsc{CCPInj}$$

Figure 15. (Untyped, labelled) congruence closure, tracking the evidence terms

C. Proofs about the congruence closure relation

C.1 Properties of typed congruence closure relation

This subsection gives the proofs for the results described in Sections 7.1 and 7.3. The main result is a theorem relating the typed congruence closure relation $\Gamma \vDash a = b$ with an untyped variation $\Gamma \vdash a = b$. The latter is defined in Figure 14.

***Definition* 22** (Injective labels)**.** We define the judgement F $\mathsf{injective}$ to mean that F is one of $- \to -$ or $\bullet - \to -$ or $- = -$.

***Lemma* 23** (Weakening for congruence closure)**.** If $\Gamma \vDash a = b$ and $\vdash \Gamma, \Gamma'$, then $\Gamma, \Gamma' \vDash a = b$.

Proof. Easy induction on $\Gamma \vDash a = b$. All cases except TCCAssumption are direct by the IH. □

***Lemma* 24** (Regularity for congruence closure)**.**
If $\Gamma \vDash a = b$ then $\Gamma \vdash a = b : \mathsf{Type}$.

Proof. Induction on $\Gamma \vDash a = b$. The cases are:

TCCRefl, TCCCongruence, TCCErasure These rules have a typing assumption which proves $\Gamma \vdash a = b : \mathsf{Type}$.

TCCSym,TCCTrans Direct by IH.

TCCInjRng By the IH, we get that $\Gamma \vdash (A_1 \to B_1) = (A_2 \to B_2)$. Applying kinding inversion (lemma 15) twice we find $\Gamma \vdash A_1 : \mathsf{Type}$ and $\Gamma, x : A_1 \vdash B_1 : \mathsf{Type}$, and similarly for A_2 and B_2. Since x is not free in B_1 (this is a simple type), by strengthening (lemma 14) we know $\Gamma \vdash B_1 : \mathsf{Type}$. Similarly, $\Gamma \vdash B_2 : \mathsf{Type}$. So we have $\Gamma \vdash B_1 = B_2 : \mathsf{Type}$ as required.

TCCInjDom,TCCIInjDom,TCCIInjRng,TCCInjEq Similar to the previous case.

□

We define the judgement $\Gamma \vdash^{\mathsf{L}} p : a = b$ ("p is evidence that $\Gamma \vdash a = b$") in the obvious way, by adding evidence terms to each inference rule in the definition in $\Gamma \vdash^{\mathsf{L}} a = b$. The resulting rules are shown in Figure 15. The grammar of evidence terms (which was also shown in the main paper) is as follows

$$p, q \quad ::= \quad x_{\triangleright p} \mid \mathsf{refl} \mid p^{-1} \mid p; q \mid \mathsf{inj}_i\ p \mid \mathsf{cong}_{\,A}\, p_1 \mathinner{..} p_i$$

Note that the notation $^{-1}$ (symmetry) and ; (transitivity) are simply syntactic constructors of evidence terms, as opposed to functions operating on evidence terms.

***Lemma* 25** ($\Gamma \vdash^{\mathsf{L}} p : a = b$ is deterministic)**.** If $\Gamma \vdash^{\mathsf{L}} p : a = b$ and $\Gamma \vdash^{\mathsf{L}} p : a' = b'$, then $a = a'$ and $b = b'$.

Proof. Simple induction on p. We implicitly assume that Γ only has one binding for any given variable. □

$$
\begin{array}{lcll}
p & \mapsto & p & \textsc{Null} \\
\mathsf{refl}^{-1} & \mapsto & \mathsf{refl} & \textsc{InvRefl} \\
\mathsf{refl}; p & \mapsto & p & \textsc{ReflTrans1} \\
p; \mathsf{refl} & \mapsto & p & \textsc{ReflTrans2} \\
(p; q); r & \mapsto & p; (q; r) & \textsc{TransTrans} \\
p; p^{-1} & \mapsto & \mathsf{refl} & \textsc{InvTrans1} \\
p^{-1}; p & \mapsto & \mathsf{refl} & \textsc{InvTrans2} \\
\\
p^{-1^{-1}} & \mapsto & p & \textsc{InvInv} \\
(p; q)^{-1} & \mapsto & q^{-1}; p^{-1} & \textsc{InvTrans} \\
\\
(\mathsf{cong}\ _A\ p_1 \mathrel{..} p_i)^{-1} & \mapsto & \mathsf{cong}\ _A\ {p_1}^{-1} \mathrel{..} {p_i}^{-1} & \textsc{InvCong} \\
\\
(\mathsf{inj}_i\ p)^{-1} & \mapsto & \mathsf{inj}_i\ (p^{-1}) & \textsc{InvInj} \\
\\
(\mathsf{cong}\ _A\ p_1 \ldots p_i); (\mathsf{cong}\ _A\ q_1 \ldots q_i) & \mapsto & \mathsf{cong}\ _A\ (p_1; q_1) \mathrel{..} (p_i; q_i) & \textsc{CongTrans} \\
\\
\mathsf{inj}_k\ (\mathsf{cong}\ _A\ p_1 \ldots p_i) & \mapsto & p_k & \textsc{InjCong1} \\
\mathsf{inj}_k\ ((\mathsf{cong}\ _A\ p_1 \mathrel{..} p_i); r) & \mapsto & p_k; (\mathsf{inj}_k\ r) & \textsc{InjCong2} \\
\mathsf{inj}_k\ (r; (\mathsf{cong}\ _A\ p_1 \ldots p_i)) & \mapsto & (\mathsf{inj}_k\ r); p_k & \textsc{InjCong3} \\
x_{\triangleright(r;\mathsf{cong}\ =\ p\ q)} & \mapsto & p^{-1}; (x_{\triangleright r}); q & \textsc{AssumCong}
\end{array}
$$

$$
\frac{p \mapsto p'}{x_{\triangleright p} \mapsto x_{\triangleright p'}}\,\textsc{Assumption}
\qquad
\frac{p \mapsto p' \quad q \mapsto q'}{p; q \mapsto p'; q'}\,\textsc{Trans}
\qquad
\frac{\forall k.\ \ p_k \mapsto p'_k}{\mathsf{cong}\ _A\ p_1 \mathrel{..} p_i \mapsto \mathsf{cong}\ _A\ p'_1 \mathrel{..} p'_i}\,\textsc{Cong}
\qquad
\frac{p \mapsto p'}{\mathsf{inj}_k\ p \mapsto \mathsf{inj}_k\ p'}\,\textsc{Inj}
$$

Figure 16. Simplification rules for evidence terms (with names for rules)

The evidence simplification relation $p \mapsto q$ was already shown in Figure 13, but we repeat it in Figure 16 in order to give names to the rules so we can conveniently refer to them. We write $\mapsto^*$ for the transitive closure of $\mapsto$.

***Lemma* 26.** If $\Gamma \vdash^{\mathsf{L}} p : a = b$ and $p \mapsto q$, then $\Gamma \vdash^{\mathsf{L}} q : a = b$.

Proof. Induction on on $p \mapsto q$, then do inversion on the assumption $\Gamma \vdash^{\mathsf{L}} p : a = b$.

In InvTrans1 and InvTrans2, we use lemma 25 to know that the two occurrences of p prove the same equation. □

Next, we define a syntactic class of of *fully simplified evidence term*, as follows.We define grammars for *synthesizable* term pS, *checkable* terms pC, and *chained* terms p^* (containing zero or more ps—an empty chain denotes the term refl, and a nonempty chain denotes a sequence of right-associated uses of transitivity $p_1; (p_2; (\ldots; p_n)))$. The metavariable p^*_{LR} ranges over chains that begin and end with a synthesizable term (as opposed to an empty chain or a chain with a pC at the beginning or end), and p^*_{R} over chains that end with a pS (but may have a pC at the beginning). Finally, x^o is an abbreviation for $x^o_{\triangleright\mathsf{refl}}$.

$$
\begin{array}{lcl}
o & ::= & 1 \mid -1 \\
pS & ::= & x^o \mid x^o_{\triangleright p^*_{\mathsf{R}}} \mid \mathsf{inj}\ i\ pS \mid p^*_{\mathsf{LR}} \\
pC & ::= & \mathsf{cong}\ _A\ p^*_1 \mathrel{..} p^*_i \\
p^* & ::= & (pS \mid pC)^* \\
p^*_{\mathsf{R}} & ::= & (p^*; pS) \\
p^*_{\mathsf{LR}} & ::= & pS \mid (pS; p^*; pS)
\end{array}
$$

There is one additional condition which is not shown in the grammar: there must never be two check-terms adjacent to each other in a chain.

***Lemma* 27.** If $\Gamma \vdash^{\mathsf{L}} p : a = b$, then there exists some p^* such that $p \mapsto^* p^*$.

Proof. As a first step, we use the $\textsc{Inv}^*$ rules to push uses of symmetry to the leaves of the evidence term. That is, the symmetry rule is only applied the uses of assumptions from the contexts. So without loss of generality we can assume that the evidence term p belongs to the following subgrammar.

$$
p \quad ::= \quad x^o_{\triangleright p} \mid \mathsf{refl} \mid p; q \mid \mathsf{inj}_i\ p \mid \mathsf{cong}\ _A\ p_1 \mathrel{..} p_i
$$

Next, we proceed by induction on the structure of p. In each case, we must show there exists some evidence chain p^* such that $p \mapsto^* p^*$.

- The term is $x^o_{\triangleright p}$. By IH, we know $p \mapsto^* p^*$.

  If p^* is empty (refl) or ends with a synthesizable term, then the term $x^o_{\triangleright p^*}$ is a valid chain and we are done.

Otherwise, p^* ends with a use of cong, i.e. p^* is $r^*; \mathsf{cong}\ _A\ q_1\ ..\ q_i$. However, by the assumption we know that $\Gamma \vdash^{\mathrm{L}} x^o_{\triangleright(r^*;\mathsf{cong}\ _A\ q_1\ ...\ q_i)} : a = b$. Assuming (wlog) that $o = 1$, this means that $\Gamma \vdash^{\mathrm{L}} (r^*; \mathsf{cong}\ _A\ q_1\ ...\ q_i) : A = (a = b)$. By inversion we know that the label A is $=$ and there are exactly two subterms q_1 and q_2, so we can simplify using AssumCong:

$$x_{\triangleright(r^*;\mathsf{cong}\ _=\ q_1\ q_2)} \mapsto {q_1}^{-1}; (x_{\triangleright r^*}); q_2$$

which is a valid chain. Similarly, in the case $o = -1$ we can simplify using AssumCong and InvTrans:

$$x^{-}_{\triangleright(r^*;\mathsf{cong}\ _=\ q_1\ q_2)}1 \mapsto {q_2}^{-1}; (x_{\triangleright r^*})^{-1}; q_1$$

- The term is refl. This is already a valid (empty) chain.

- The term is $p; q$. By the IHs for p and q we know that there are chains p^* and q^*. We must now show that $p^*; q^*$ can be simplified into a valid chain r^*.

  If p^* is the empty chain refl, then by ReflTrans1 we can just return q^*. Similarly if q^* is empty, then by ReflTrans2 we can return p^*.

  If both p^* and q^* are nonempty, we use TransTrans to reassociate $p^*; q^*$ into a right-associated chain. However, we must also ensure that the resulting chain does not contain two adjacent pCs. That would happen if p^* ends with a use of cong and q^* begins with cong. In that case, after reassociation we end up with a subproof of the form

$$(\mathsf{cong}\ _A\ p_1\ ..\ p_i); (\mathsf{cong}\ _B\ q_1\ ..\ q_j)$$

  By assumption we know this is evidence for some equation $a = b$. By inversion on the judgement

$$\Gamma \vdash^{\mathrm{L}} (\mathsf{cong}\ _A\ p_1\ ..\ p_i); (\mathsf{cong}\ _B\ q_1\ ..\ q_j) : a = b$$

  we see that we must have $A = B$ and $i = j$, and $a = b$ must be $_A\ \overline{a_i} =_A\ \overline{b_i}$. Then we can use CongTrans to simplify to a single use of cong.

- The term is $\mathsf{inj}_i\ p$. By IH we know $p \mapsto^* p^*$.

  Now, $\mathsf{inj}\ i\ p^*$ may not be a valid normalized evidence term, because it may violate the condition that p^* begins and ends with a pS. Let $p^* = q_1^*; q_2^*; q_3^*$, such that q_1^* and q_3^* consists only of checkable terms and q_2^* begins and ends with a synthesizable term. Now apply InjCong2 and InjCong3 repeatedly to simplify q_1^* and q_3^*. We get

$$\mathsf{inj}\ i\ p^* \mapsto^* r_1^*; (\mathsf{inj}\ i\ q_2^*); r_3^*$$

  where r_1^* consists of subterms from the cong-expressions in q_1^*, and similarly for q_3^*.

  Finally, at this point r_1^* and r_3^* may contain adjacent cong-terms, so we need to simplify them using CongTrans as in the previous case.

- The term is $\mathsf{cong}\ _A\ p_1\ ..\ p_i$. By the IHs, we know $p_k \mapsto^* p_k^*$. Then

$$\mathsf{cong}\ _A\ p_1\ ..\ p_i \mapsto^* \mathsf{cong}\ _A\ p_1^*\ ..\ p_i^*$$

  which is a valid chain.

□

Intuitively, the label function recursively decomposes a term a into a first-ordered "labelled" expression $F(a_1, \ldots, a_k)$, where F is the least nontrival linear multi-hole context that agrees with a. The label function takes an expression a, and returns a label F together with a list of subexpressions ak. We write this as

$$\mathsf{label}\ a = F\ \overline{a_i}$$

The function label is defined in turns of a helper function $\mathsf{label}_S\ a$, which takes as argument a set of variables S and an expression a and also returns $_A\ \overline{a_i}$, with the additional constraint that label_S tries to select the smallest label F such that $\mathsf{FV}\,(a_k) \cap S = \emptyset$. The two functions are quite similar (in the Haskell implementation there is just one function which takes an extra boolean argument); the difference is that label_S

can return the trivial context which is just a single hole, whereas label always chooses a label that contains at least one syntactic constructor.

$$
\begin{array}{lcl}
\mathsf{label}\,\mathsf{Type} & = & (\mathsf{Type}) \\
\mathsf{label}\,x & = & (x) \\
\mathsf{label}\,(\mathsf{rec}\ f_A\ x.a) & = & (\mathsf{rec}\ f\ x.F)\ \overline{a_i} \\
\multicolumn{3}{c}{\text{where } \mathsf{label}_{\{f,x\}}\ a = F\ \overline{a_i}} \\
\mathsf{label}\,(\mathsf{rec}\ f_A\ \bullet_x.a) & = & (\mathsf{rec}\ f\ \ \bullet.F)\ \overline{a_i} \\
\multicolumn{3}{c}{\text{where } \mathsf{label}_{\{f,x\}}\ a = F\ \overline{a_i}} \\
\mathsf{label}\,(a\ b) & = & (-\,-)\ (\mathsf{label}\ a)\ (\mathsf{label}\ b) \\
\mathsf{label}\,(a\ \bullet_b) & = & (-\ \bullet)\ (\mathsf{label}\ a) \\
\mathsf{label}\,((x\!:\!A) \to\ B) & = & (x:-) \to F)\ (\mathsf{label}\ A)\ \overline{B_i} \\
\quad \text{where } \mathsf{label}_{\{x\}}\ B = F\ \overline{B_i} & & \\
\mathsf{label}\,(\bullet(x\!:\!A) \to\ B) & = & \bullet(x:-) \to F)\ (\mathsf{label}\ A)\ \overline{B_i} \\
\quad \text{where } \mathsf{label}_{\{x\}}\ B = F\ \overline{B_i} & & \\
\mathsf{label}\,(a = b) & = & (- = -)\ (\mathsf{label}\ a)\ (\mathsf{label}\ b) \\
\mathsf{label}\,\mathsf{join}_\Sigma & = & (\mathsf{join}) \\
\mathsf{label}\,(a_{\rhd b}) & = & \mathsf{label}\ a
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{label}_S\ a & = & (-)\ (\mathsf{label}\ a) \\
\quad \text{when } \mathsf{FV}\,(a) \cap S = \emptyset & & \\
\text{Otherwise:} & & \\
\mathsf{label}_S\ x & = & (x) \\
\mathsf{label}_S\ (\mathsf{rec}\ f_A\ x.a) & = & (\mathsf{rec}\ f\ x.F)\ \overline{a_i} \\
\quad \text{where } \mathsf{label}_{S\cup\{f,x\}}\ a = F\ \overline{a_i} & & \\
\mathsf{label}_S\ (\mathsf{rec}\ f_A\ \bullet_x.a) & = & (\mathsf{rec}\ f\ \ \bullet.F)\ \overline{a_i} \\
\quad \text{where } \mathsf{label}_{S\cup\{f,x\}}\ a = F\ \overline{a_i} & & \\
\mathsf{label}_S\ (a\ b) & = & (F\ G)\ \overline{a_i}\ \overline{b_i} \\
\quad \text{where } \mathsf{label}_S\ a = F\ \overline{a_i} & & \\
\quad \text{and} \quad \mathsf{label}_S\ b = G\ \overline{b_i} & & \\
\mathsf{label}_S\ (a\ \bullet_b) & = & (F\ \bullet)\ \overline{a_i} \\
\quad \text{where } \mathsf{label}_S\ a = F\ \overline{a_i} & & \\
\mathsf{label}_S\ ((x\!:\!A) \to\ B) & = & ((x:F) \to G)\ \overline{A_i}\ \overline{B_i} \\
\quad \text{where } \mathsf{label}_S\ A = F\ \overline{A_i} & & \\
\quad \text{and} \quad \mathsf{label}_{S\cup\{x\}}\ B = G\ \overline{B_i} & & \\
\mathsf{label}_S\ (\bullet(x\!:\!A) \to\ B) & = & (\bullet(x:F) \to G)\ \overline{A_i}\ \overline{B_i} \\
\quad \text{where } \mathsf{label}_S\ A = F\ \overline{A_i} & & \\
\quad \text{and} \quad \mathsf{label}_{S\cup\{x\}}\ B = G\ \overline{B_i} & & \\
\mathsf{label}_S\ (a = b) & = & (F\ G)\ \overline{a_i}\ \overline{b_i} \\
\quad \text{where } \mathsf{label}_S\ a = F\ \overline{a_i} & & \\
\quad \text{and} \quad \mathsf{label}_S\ b = G\ \overline{b_i} & & \\
\mathsf{label}_S\ \mathsf{join}_\Sigma & = & (\mathsf{join}) \\
\mathsf{label}_S\ (a_{\rhd b}) & = & \mathsf{label}_S\ a
\end{array}
$$

We also define the "inverse" function unlabel, which simply substitutes away all the label applications. The function unlabel is defined by recursion on the labelled term:

$$\mathsf{unlabel}\,(F\ \overline{a_i}) = \{\mathsf{unlabel}\ a_1/x_1\}\ ...\ \{\mathsf{unlabel}\ a_j/x_j\}F$$

when the holes in F are named x_1 through x_j.

Lemmas 28–31 are all proved by inductions on the term a.

***Lemma* 28** (unlabel-label)**.** For any a, we have $\mathsf{unlabel}\,(\mathsf{label}\ a) = |a|$.

***Lemma* 29** (Substituting into a label)**.** Suppose $\mathsf{label}\ a' = F\ \overline{a_i}$ where the holes in F are named $x_1 \ldots x_i$. Then $|a'| = |\{\mathsf{unlabel}\ a_1/x_1\}\ ...\ \{\mathsf{unlabel}\ a_j/x_k\}\ F$

***Lemma* 30** (label does not let bound variables escape)**.**

- If $\mathsf{label}\ a = F\ \overline{a_i}$, then for every k we have $\mathsf{FV}\,(a_k) \subseteq \mathsf{FV}\,(a)$.
- If $\mathsf{label}_S\ a = F\ \overline{a_i}$, then for every k we have $\mathsf{FV}\,(a_k) \subseteq (\mathsf{FV}\,(a) \setminus S)$.

***Lemma* 31** (label decides erasure)**.** For any expressions a and b, we have $|a| = |b|$ iff $(\mathsf{label}\ a) = (\mathsf{label}\ b)$

***Lemma* 32.** For all a, b and c such that $\mathsf{FV}\,(a) \cap S = \emptyset$ and $\mathsf{FV}\,(b) \cap S = \emptyset$, if $\mathsf{label}_S\ \{a/x\}\ c = F\ \overline{a_i}$ and $\mathsf{label}_S\ \{b/x\}\ c = G\ \overline{b_i}$, then $F = G$, and there exists $\overline{c_i}$ such that for all k, $a_k = \mathsf{label}\,\{a/x\}\ c_k$ and $b_k = \mathsf{label}\,\{b/x\}\ c_k$.a

Proof. Induction on the structure of c.

c **is** x Then since we assumed that a and b have no free variables in S, $\mathsf{label}_S\ \{a/x\}\ c = (–)\ (\mathsf{label}\ a)$ and $\mathsf{label}_S\ \{b/x\}\ c = (–)\ (\mathsf{label}\ b)$, so the labels are equal and we can take the list to be just $c_0 = x$.

c **is** Type Then $\mathsf{label}_S\ \{a/x\}\ c = \mathsf{label}_S\ \{b/x\}\ c = (\mathsf{Type})$, so the labels are indeed equal, and we can take the empty list for $\overline{c_i}$.

c **is some variable** $y \neq x$ Similar to the previous case.

c **is** join_Σ Similar to the previous case.

c **is** $\mathsf{rec}\ f_A\ y.c_0$ Let $\mathsf{label}_S\ \{a/x\}\ c_0 = F\,\overline{a_i}$ and $\mathsf{label}_S\ \{b/x\}\ c_0 = G\,\overline{b_i}$. By the IH we know $F = G$, and there is a list $\overline{c_i}$.

Now, $\mathsf{label}_S\ \{a/x\}\ c = (\ \mathsf{rec} f\ y.F)\ \overline{a_i}$ and $\mathsf{label}_S\ \{b/x\}\ c = (\ \mathsf{rec} f\ y.G)\ \overline{b_i}$. So the labels are indeed equal, and the list of expressions is just $\overline{c_i}$.

c **is** $\mathsf{rec}\ f_A\ \bullet_y.c_0$ Similar to the previous case.

c **is** $(y : C_1) \rightarrow\ C_2$ Let

$$\begin{array}{lcl} \mathsf{label}_S\ \{a/x\}\ C_1 & = & F\,\overline{a_i} \\ \mathsf{label}_S\ \{b/x\}\ C_1 & = & G\,\overline{b_i} \\ \mathsf{label}_{S\cup\{y\}}\ \{a/x\}\ C_2 & = & F'\,\overline{a_i}' \\ \mathsf{label}_{S\cup\{y\}}\ \{b/x\}\ C_2 & = & G'\,\overline{b_i}' \end{array}$$

Since we can choose the bound variable y fresh, the disjointness condition on S is still satisfied. So by the IHs we get that $F = G$ and $F' = G'$, and also suitable lists $\overline{c_i}$ and $\overline{c_i}$.

Now, $\mathsf{label}_S\ \{a/x\}\ ((y : C_1) \rightarrow\ C_2) = ((y : F) \rightarrow F')\ \overline{a_i}\ \overline{a_i}'$ and $\mathsf{label}_S\ \{b/x\}\ ((y : C_1) \rightarrow\ C_2) = ((y : F) \rightarrow F')\ \overline{b_i}\ \overline{b_i}'$. So the label is indeed the same for both applications, and $\overline{c_i}\ \overline{c_i}'$ is a suitable list.

c **is** $\bullet(y : C_1) \rightarrow\ C_2$ **or** $c_1\ c_2$ **or** $c_1 = c_2$ Similar to the previous case.

c **is** $c_1\ \bullet_{c_2}$ Let

$$\begin{array}{l} \mathsf{label}_S\ \{a/x\}\ c_1 = F\,\overline{a_i} \\ \mathsf{label}_S\ \{b/x\}\ c_1 = G\,\overline{b_i} \end{array}$$

The IH gives $F = G$ and a list $\overline{c_i}$. The label we return is $(F\ \bullet)$, and the argument list is $\overline{c_i}$.

c **is** $c_{1 \triangleright c_2}$ Similar to the previous case.

$\square$

***Lemma* 33** (CC implies LCC, the congruence case)**.** For all c, if $\Gamma \vdash^{\mathsf{L}} \mathsf{label}\ a = \mathsf{label}\ b$ then $\Gamma \vdash^{\mathsf{L}} \mathsf{label}\,\{a/x\}\ c = \mathsf{label}\,\{b/x\}\ c$

Proof. Simultaneous induction on the structure of c. Most of the cases are similar, so we show only some representative ones.

c **is** x we must show $\Gamma \vdash^{\mathsf{L}} \mathsf{label}\ a = \mathsf{label}\ b$, which we have as an assumption.

c **is** Type . Then both $\mathsf{label}\,\{a/x\}\ c$ and $\mathsf{label}\,\{b/x\}\ c$ are just (Type), so LCCREFL proves the required equation.

c **is a variable** $y \neq x$ Similar to previous case.

c **is some variable** join_Σ Similar to the previous case.

c **is** $\mathsf{rec}\ f_A\ y.c_0$ By lemma 32, there is some F and $\overline{c_i}$ such that $\mathsf{label}_{\{f,y\}}\ \{a/x\}\ c_0 = F\,\overline{a_i}$ and $\mathsf{label}_{\{f,y\}}\ \{b/x\}\ c_0 = F\,\overline{b_i}$, and furthermore $a_k = \mathsf{label}\,\{a/x\}\ c_k$ and $b_k = \mathsf{label}\,\{b/x\}\ c_k$.

By the IH, we know that $\forall k.\ \ \Gamma \vdash^{\mathsf{L}} a_k = b_k$.

But note that $\mathsf{label}\,\{a/x\}\,(\mathsf{rec}\ f_A\ y.c_0)$ is $(\ \mathsf{rec} f\ y.F)\ \overline{a_i}$, and $\mathsf{label}\,\{b/x\}\,(\mathsf{rec}\ f_A\ y.c_0)$ is $(\ \mathsf{rec} f\ y.F)\ \overline{b_i}$. So by LCCCONG using the label $(\ \mathsf{rec} f\ y.F)$ we have $\Gamma \vdash^{\mathsf{L}} \{a/x\}\ c = \{b/x\}\ c$ as required.

c **is** $\mathsf{rec}\ f_A\ \bullet_y.c_0$ Similar to the previous case.

c **is** $(y : C_1) \rightarrow\ C_2$ By lemma 32, there is some F and $\overline{c_i}$ such that $\mathsf{label}_S\ \{a/x\}\ C_2 = F\,\overline{a_i}$ and $\mathsf{label}_S\ \{b/x\}\ C_2 = F\,\overline{b_i}$, and furthermore $a_k = \mathsf{label}\,\{a/x\}\ c_k$ and $b_k = \mathsf{label}\,\{b/x\}\ c_k$.

Now $\mathsf{label}\,\{a/x\}\ c = ((y : –) \rightarrow F)\ (\mathsf{label}\,\{a/x\}\ C_1)\overline{a_i}$. and $\mathsf{label}\,\{b/x\}\ c = ((y : –) \rightarrow F)\ (\mathsf{label}\,\{b/x\}\ C_1)\overline{b_i}$.

By the IHs we get $\Gamma \vdash^{\mathsf{L}} \mathsf{label}\,\{a/x\}\ C_1 = \mathsf{label}\,\{b/x\}\ C_1$ and also $\forall k.\ \ \Gamma \vdash^{\mathsf{L}} a_k = b_k$. So we conclude by LCCCONG using the label $((y : –) \rightarrow F)$.

c **is** $\bullet(y : C_1) \rightarrow\ C_2$ **or** $c_1\ c_2$ **or** $c_1 = c_2$ Similar to the previous case.

c **is** $c_1\ \bullet_{c_2}$ Then $\mathsf{label}\,\{a/x\}\ c = (–\ \bullet)\ (\mathsf{label}\,\{a/x\}\ c_1)$ and $\mathsf{label}\,\{b/x\}\ c = (–\ \bullet)\ (\mathsf{label}\,\{b/x\}\ c_1)$. By the IH we have $\Gamma \vdash^{\mathsf{L}} \mathsf{label}\,\{a/x\}\ c_1 = \mathsf{label}\,\{b/x\}\ b_1$. So we conclude by LCCCONG using the label $(–\ \bullet)$.

c **is** $c_{1 \triangleright c_2}$ Similar to the previous case.

□

***Lemma* 34** (label preserves CC)**.** If $\Gamma \vdash a = b$, then $\mathsf{label}\,\Gamma \vdash^{\mathrm{L}} \mathsf{label}\; a = \mathsf{label}\; b$.

Proof. Induction on $\Gamma \vdash a = b$. The cases are

CCREFL We are given

$$\frac{|a| = |b|}{\Gamma \vdash a = b}\textsc{CCRefl}$$

From $|a| = |b|$ and lemma 31 we know $\mathsf{label}\; a = \mathsf{label}\; b$. So apply LCCREFL.

CCSYM,CCTRANS These follow directly by IH.

CCASSUMPTION We are given

$$\frac{x : A \in \Gamma \quad \Gamma \vdash A = (a = b)}{\Gamma \vdash a = b}\textsc{CCAssumption}$$

Since $x : A \in \Gamma$ we know $x : \mathsf{label}\; A \in \mathsf{label}\,\Gamma$. And by the IH we have $\mathsf{label}\,\Gamma \vdash \mathsf{label}\; A = \mathsf{label}\,(a = b)$. Since $\mathsf{label}\,(a = b)$ is the same as $\mathsf{label}\; a = \mathsf{label}\; b$, we conclude by LCCASSUMPTION.

CCCONGRUENCE We are given

$$\frac{\Gamma \vdash a = b}{\Gamma \vdash \{a/x\}\; c = \{b/x\}\; c}\textsc{CCCongruence}$$

Apply lemma 33.

CCINJDOM We are given

$$\frac{\Gamma \vdash (A_1 \to B_1) = (A_2 \to B_2)}{\Gamma \vdash A_1 = B_1}\textsc{CCInjDom}$$

The IH gives $\mathsf{label}\,\Gamma \vdash \mathsf{label}\,(A_1 \to B_1) = \mathsf{label}\,(A_2 \to B_2)$, which is the same as $\mathsf{label}\,\Gamma \vdash (- \to -)\; (\mathsf{label}\; A_1)\; (\mathsf{label}\; B_1) = (- \to -)\; (\mathsf{label}\; A_2)\; (\mathsf{label}\; B_2)$. And $(- \to -)$ is an injective label, so we conclude by CCINJECTIVITY.

CCINJRNG,CCIINJDOM,CCIINJRNG,CCINJEQ These cases are similar to the previous one.

□

***Lemma* 35** (Label arguments arise from well-typed subexpressions)**.**
- If $\Gamma \vdash a' : A$, and $\mathsf{label}\; a' = F\; \overline{a_i}$, then for every a_k there exists a'_k such that $\Gamma \vdash a'_k : A_k$ and $a_k = \mathsf{label}\; a'_k$.
- If $\Gamma \vdash a' : A$, and $\mathsf{label}_S\; a' = F\; \overline{a_i}$, then for every a_k there exists a'_k such that $\Gamma \vdash a'_k : A_k$ and $a_k = \mathsf{label}\; a'_k$.

Proof. (Strong) induction on the structure of a'. Most of the cases of the induction are similar, so we do not show all of them. A few representative cases for label are:

a' is Type or some variable x Then $\mathsf{label}\; a'$ is a nullary label-application, so $\overline{a_i}$ is empty and the lemma is vacuously true.

a' is $\mathsf{rec}\; f_A\; x.b'$ There is only one typing rule for rec-expressions, so from the judgement $\Gamma \vdash a' : A$, we know that

$$\Gamma, f : (x : A_1) \to A_2, x : A_1 \vdash b' : A_2$$

From the definition of label we know that $\mathsf{label}\; a'$ is $(\mathsf{rec}\; f_{:A}\; x.F)\; \overline{a_i}$, where $\mathsf{label}_{\{f,x\}}\; b = F\; \overline{a_i}$. So by the IH for a' we know that there exists a'_k such that $a_k = \mathsf{label}\; a'_k$ and $\Gamma, f : (x : A_1) \to A_2, x : A_1 \vdash a'_k : A'$. By lemma 30 we know that f and x are not free in a'_k, so by strengthening (lemma 14) we have $\Gamma \vdash a'_k : A'$ as required.

a' is $b'\; c'$ Then $\mathsf{label}\; a'$ is $(-\; -)\; \overline{b_i}\; \overline{c_i}$. The expression a_k must belong to one of the lists $\overline{b_i}$ or $\overline{c_i}$, so by the IH for b' or c' we get a corresponding b'_k or c'_k.

A few representative cases for label_S are:

a' has no free variables in S Then $\mathsf{label}_S\; a' = (-)\; (\mathsf{label}\; a')$. So there is only a single a_k, which must be $(\mathsf{label}\; a')$. Thus we can take $a'_k = a'$.

a' is Type or some variable x Similar to the corresponding case for label: $\mathsf{label}_S\; a'$ is a nullary label-application and the lemma is vacuously true.

a' is $\mathsf{rec}\; f_A\; x.b'$ As in the case for label, we know that

$$\Gamma, f : (x : A_1) \to A_2, x : A_1 \vdash b' : A_2$$

and $\mathsf{label}_S\; a'$ is $(\mathsf{rec}\; f_{:A}\; x.F)\; \overline{a_i}$, where $\mathsf{label}_{S \cup \{f,x\}}\; b = F\; \overline{a_i}$. Conclude by IH and strengthening as in the above case.

a' is $b'\; c'$ Then $\mathsf{label}_S\; a'$ is $(F\; G)\; \overline{b_i}\; \overline{c_i}$. The expression a_k must belong to one of the lists $\overline{b_i}$ or $\overline{c_i}$, so by the IH for b' or c' we get a corresponding b'_k or c'_k.

$\square$

***Lemma* 36** (Inversion for label). • If $(\mathsf{label}\ A') = (- = -)\ a\ b$, then there exists a' and b' such that $A' = (a' = b')$ and $(\mathsf{label}\ a') = a$ and $(\mathsf{label}\ b') = b$.

- If $\mathsf{label}\ A' = (- \to -)\ a_1\ a_2$, then there exists a_1' and a_2' such that $A' = (a_1' \to a_2')$, and $(\mathsf{label}\ a_1') = a_1$ and $(\mathsf{label}\ a_2') = a_2$.
- Similar for $\bullet a_1' \to\ a_2'$
- Similar fpr $a_1' = a_2'$.

Proof. Immediate from considering cases for A' and examining the definition of label. $\square$

***Lemma* 37** (Normalized untyped CC implies typed CC). • If $\mathsf{label}\,\Gamma' \vdash^{\mathsf{L}} pS : a = b$, then there exists annotated core expressions a', b' such that $a = \mathsf{label}\ a'$ and $b = \mathsf{label}\ b'$ and $\Gamma' \vDash a' = b'$.

- If $\mathsf{label}\,\Gamma' \vdash^{\mathsf{L}} pC : \mathsf{label}\ a' = \mathsf{label}\ b'$ and $\Gamma' \vdash a' = b' : \mathsf{Type}$, then $\Gamma' \vDash a' = b'$.
- If $\mathsf{label}\,\Gamma' \vdash^{\mathsf{L}} p^* : \mathsf{label}\ a' = \mathsf{label}\ b'$ and $\Gamma' \vdash a' = b' : \mathsf{Type}$, then $\Gamma' \vDash a' = b'$.
- If $\mathsf{label}\,\Gamma' \vdash^{\mathsf{L}} p_{\mathsf{R}}^* : \mathsf{label}\ a' = b$ and $\Gamma' \vdash a' : A$, then there exists an b' such that $b = \mathsf{label}\ b'$ and $\Gamma' \vDash a' = b'$.

Proof. We proceed by mutual induction on the sizes of pS and p^*. The cases for pS are:

The evidence is $x_{\triangleright p_{\mathsf{R}}^*}$ By examining the definition of $\Gamma \vdash^{\mathsf{L}} p : a = b$, we see that the only rule that applies is CCPASSUMPTION, so we know we have

$$x : A \in (\mathsf{label}\,\Gamma')$$
$$\mathsf{label}\,\Gamma' \vdash^{\mathsf{L}} p_{\mathsf{R}}^* : A = ((- = -)\ a\ b)$$

From $x : A \in (\mathsf{label}\,\Gamma')$ we know that $A = \mathsf{label}\ A'$ for some $x : A' \in \Gamma'$.

Then from the mutual IH for p_{R}^* we known that there exists some B' such that $((- = -)\ a\ b) = \mathsf{label}\ B'$ and $\Gamma' \vDash A' = B'$.

Further, by lemma 36 we know that $B' = \mathsf{label}\,(a' = b')$ for some expressions a' and b' such that $a = \mathsf{label}\ a'$ and $b = \mathsf{label}\ b'$. So we have shown $\Gamma' \vDash A' = (a' = b')$. Now apply TCCASSUMPTION to conclude $\Gamma' \vDash a' = b'$ as required.

The evidence is $(x_{\triangleright p_{\mathsf{R}}^*})^{-1}$ By reasoning as in the previous case we get some a' and b' such that $a = \mathsf{label}\ a'$ and $b = \mathsf{label}\ b'$ and $\Gamma' \vDash a' = b'$. Then apply TCCSYM to conclude $\Gamma' \vDash b' = b'$ as required.

The evidence is $\mathsf{inj}\ i\ pS$ By examining the definition of the $\Gamma \vdash^{\mathsf{L}} p : a = b$ judgement we see that the only rule that applies is CCPINJ. So we must have

$$\mathsf{label}\,\Gamma' \vdash^{\mathsf{L}} pS : F\,\overline{a_i} = F\,\overline{b_i}$$
$$F\ \mathsf{injective}$$

Recall that F injective means that F is either $- \to -$, $- = -$, or $\bullet - \to\ -$.

We consider the case when it is $- \to -$ and i is 1; the other cases are similar. That is, the assumed derivation looks like

$$\frac{\mathsf{label}\,\Gamma' \vdash pS : (- \to -)\ a_1\ a_2 = (- \to -)\ b_1\ b_2}{\mathsf{label}\,\Gamma' \vdash^{\mathsf{L}} \mathsf{inj}\ i\ pS : a_1 = b_1}$$

From the IH we get expressions A' and B' such that $(- \to -)\ a_1\ a_2 = \mathsf{label}\ A'$ and $(- \to -)\ b_1\ b_2 = \mathsf{label}\ B'$, and $\Gamma' \vDash A' = B'$. By lemma 36 we then know $A' = (a_1' \to a_2')$ and $B' = (b_1' \to b_2')$ Then apply TCCINJDOM to conclude $\Gamma' \vDash a_1' = b_1'$ as required.

The evidence is a chain p_{LR}^* From the grammar for p_{LR}^* that means that it is either a single terms pS (which we dealt with in the above cases), or it is a chain starting and ending with a synthesizable term, that is p_{LR}^* is $pS; q^*; rS$.

In the latter case, by the IH for pS and rS we get terms c_1' and c_2' such that $\Gamma \vDash \mathsf{label}\ a' = \mathsf{label}\ c_1'$ and $\Gamma \vDash \mathsf{label}\ c_1' = b'$.

Now we can apply the mutual induction hypothesis for the chain r^*, to get $\Gamma' \vDash c_1' = c_2'$.

Finally, apply transitivity (CCPTRANS) twice to conclude $\Gamma' \vDash a' = b'$ as required.

The only case for pC is when **the evidence term is a use of congruence,** $\mathsf{cong}\ F\ p_1 .. p_i$. The only rule that applies is CCPCONG, so the assumed derivation is

$$\frac{\forall k.\ \ \mathsf{label}\,\Gamma' \vdash^{\mathsf{L}} p_k : a_k = b_k}{\mathsf{label}\,\Gamma' \vdash^{\mathsf{L}} \mathsf{cong}\ F\ p_1 .. p_i : F\,\overline{a_i} = F\,\overline{b_i}}$$

By assumption we know that $(F\,\overline{a_i}) = (\mathsf{label}\ a')$ and $(F\,\overline{b_i}) = (\mathsf{label}\ b')$.

From the assumption $\Gamma \vdash a' = b' : \mathsf{Type}$ we know a' and b' are well typed, so by lemma 35 we know that for every a_k there exists a well typed a_k' such that $a_k = \mathsf{label}\ a_k'$, and similarly for b_k.

So from IH for p_k we know $\forall k.\ \ \Gamma' \vDash a_k' = b_k'$.

By lemma 29 we know that $|a'| = |\{\mathsf{unlabel}\ a_1/x_1\} \ldots \{\mathsf{unlabel}\ a_i/x_i\}\ F|$. Since $\mathsf{unlabel}$ is inverse to label (lemma 28) this means $|a'| = |\{a'_1/x_1\} \ldots \{a'_i/x_i\}\ F|$. Similarly, $|b'| = |\{b'_1/x_1\} \ldots \{b'_i/x_i\}\ F|$. Finally, we know that $\Gamma' \vdash a' = b' : \mathsf{Type}$ by the assumption to the theorem.

So by TCCCONGRUENCE, $\Gamma' \vDash a' = b'$ as required.

The cases for p^* are:

The empty chain (refl) The only rule that can apply is CCPREFL, so we know that $(\mathsf{label}\ a') = (\mathsf{label}\ b')$. By lemma 31 this implies that $|a'| = |b'|$. We know as an assumption to the lemma that $\Gamma' \vdash a' = b'$, apply TCCERASURE to conclude $\Gamma' \vDash a' = b'$ as required.

A chain consisting of a single term, p The evidence term p must be either a checkable are a synthesizable term. In the case when it is a pC we directly appeal to the mutual IH.

In the case when it is a pS, by the mutual IH we know that there are a'' and b'' such that $a = \mathsf{label}\ a''$ and $b = \mathsf{label}\ b''$ and $\mathsf{label}\ \Gamma' \vDash a'' = b''$.

Since $\mathsf{label}\ a' = \mathsf{label}\ a''$, by lemma 31 we know $|a'| = |a''|$, and similarly $|b''| = |b'|$. So by two uses each of TCCERASURE and TCCTRANS we get $\mathsf{label}\ \Gamma' \vDash a' = b'$, as required.

A chain of length > 1, starting with synthesizable term, $pS; q^*$ The only rule that applies is CCPTRANS, so we must have

$$\begin{array}{l}\mathsf{label}\ \Gamma' \vdash^{\mathsf{L}} p : a = c \\ \mathsf{label}\ \Gamma' \vdash^{\mathsf{L}} q^* : c = b\end{array}$$

From the mutual IH for pS we know that there is some a'' and c'' such that $a = \mathsf{label}\ a''$, $c = \mathsf{label}\ c''$, and $\mathsf{label}\ \Gamma' \vDash a'' = c''$. By reasoning as in the previous case we also know that $\mathsf{label}\ \Gamma' \vDash a' = a''$.

Now by the IH for q^* we know $\mathsf{label}\ \Gamma' \vDash c'' = b'$.

So by transitivity (TCCTRANS) we get $\mathsf{label}\ \Gamma' \vDash a' = b''$ as required.

A chain of length > 1, starting with a checkable term, $pC; qS; r^*$ The definition of chains stipulates that there must never be two adjacent pCs, so we know that the second evidence term in the chain, qS, is synthesizable.

The only rule that applies is CCPTRANS, so we must have

$$\begin{array}{l}\mathsf{label}\ \Gamma' \vdash^{\mathsf{L}} pC : a = c_1 \\ \mathsf{label}\ \Gamma' \vdash^{\mathsf{L}} qS : c_1 = c_2 \\ \mathsf{label}\ \Gamma' \vdash^{\mathsf{L}} r^* : c_2 = b\end{array}$$

By the mutual IH for qS we get suitable c'_1 and c'_2. Then apply the IHs for pC and r^*.

The cases for p^*_{R} are similar to the reasoning for general chains p^*. ☐

***Lemma* 38** (Core proof terms for $\Gamma \vDash a = b$)**.** If $\Gamma \vDash a = b$, then there exists some value v in the annotated core language such that $\Gamma \vdash v : a = b$.

Proof. Induction on the judgement $\Gamma \vDash a = b$.

TCCERASURE The assumed derivation looks like

$$\frac{|a| = |b| \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vDash a = b}\ \text{TCCERASURE}$$

From the regularity assumptions $\Gamma \vdash a : A$ and $\Gamma \vdash b : B$ we know $\Gamma \vdash a = b : \mathsf{Type}$. So the equation follows from a use of join:

$$\frac{|a| \leadsto^{0}_{\mathsf{cbv}} |a| \quad |b| \leadsto^{0}_{\mathsf{cbv}} |a| \quad \Gamma' \vdash a = b : \mathsf{Type}}{\Gamma' \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} 0 0 : a = b} : a = b}$$

TCCREFL Similar to the previous case.

TCCSYM By IH we get $\Gamma \vdash v : a = b$. From regularity (lemma 24) we know that a is typeable, so $\Gamma \vdash a = a : \mathsf{Type}$. then we can prove $b = b$ using TCAST, TSUBST and TJOINC, as follows:

$$\frac{\dfrac{\Gamma \vdash v : a = b}{\Gamma \vdash \mathsf{join}_{\sim v = a} : (a = a) = (b = a)} \quad \dfrac{\Gamma \vdash a = a : \mathsf{Type}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} 0 0 : a = a} : a = a}}{\Gamma \vdash \mathsf{join}_{\leadsto_{\mathsf{cbv}} 0 0 : a = a \triangleright \mathsf{join}_{\sim v = a}} : b = a}$$

TCCTRANS The assumed derivation looks like

$$\frac{\Gamma \vDash a = b \quad \Gamma \vDash b = c}{\Gamma \vDash a = c}\ \text{TCCTRANS}$$

The IHs are $\Gamma \vdash v_1 : a = b$ and $\Gamma \vdash v_2 : b = c$. We can then prove $a = c$ using TJCAST and TJSUBST:

$$\dfrac{\Gamma \vdash v_1 : a = b \quad \dfrac{\Gamma \vdash v_2 : b = c}{\Gamma \vdash \mathsf{join}_{a=\sim v_2} : (a = b) = (a = c)}}{\Gamma \vdash v_{\triangleright \mathsf{join}_{a=\sim v_2}} : a = c}$$

TCCASSUMPTION The assumed derivation looks like

$$\dfrac{x : A \in \Gamma \quad \Gamma \vDash A = (a = b)}{\Gamma \vDash a = b}\text{TCCASSUMPTION}$$

The IH gives $\Gamma \vdash v : A = (a = b)$, so $\Gamma \vdash x_{\triangleright v} : a = b$.

TCCCONGRUENCE The assumed derivation looks like

$$\dfrac{\begin{array}{c}\Gamma \vdash A = B : \mathsf{Type} \quad \forall k.\ \Gamma \vDash a_k = b_k \\ |A = B| = |\{a_1/x_1\} \ldots \{a_j/x_j\}\, c = \{b_1/x_1\} \ldots \{b_j/x_j\}\, c|\end{array}}{\Gamma \vDash A = B}\text{TCCCONGRUENCE}$$

The IH gives $\overline{v_i}$ such that $\forall k.\ \Gamma \vdash v_k : a_k = b_k$. By the regularity assumption to the rule we know that the equation is well-typed. So by TJSUBST we have

$$\Gamma \vdash \mathsf{join}_{\{\sim v_1/x_1\} \ldots \{\sim v_j/x_j\}\, c : A = B} : A = B$$

as required.

TCCINJDOM From the IH we have $\Gamma \vdash v : ((x : A_1) \to B_1) = ((x : A_2) \to B_2)$. So apply TJINJDOM to get $\Gamma \vdash \mathsf{join}_{\mathsf{injdom}\ v} : A_1 = A_2$ as required.

TCCINJRNG, TCCIINJDOM, TCCIINJRANGE, TCCINJEQ Similar to the TCCINJDOM case.

□

***Theorem* 39** (Typed CC from untyped CC)**.** Suppose $\Gamma \vdash a = b$ and $\Gamma \vdash a = b : \mathsf{Type}$. Then $\Gamma \vDash a = b$, and furthermore $\Gamma \vdash v : a = b$ for some v.

Proof. From $\Gamma \vdash a = b$, by lemma 34 we get $\Gamma \vdash \mathsf{label}\, a = \mathsf{label}\, b$. By evidence simplification (lemma 27) we get $\Gamma \vdash^{\mathsf{L}} p^* : \mathsf{label}\, a = \mathsf{label}\, b$. From this, and the fact that $\Gamma \vdash a = b : \mathsf{Type}$, by lemma 37 we get $\Gamma \vDash a = b$ as required. Finally, by lemma 38 there is some v such that $\Gamma \vdash v : a = b$. □

***Lemma* 40.** If $\Gamma \vDash a = b$ then $\Gamma \vdash a = b$.

Proof. Induction on $\Gamma \vDash a = b$.

TCCREFL,TCCERASURE By CCREFL.

TCCSYM,TCCTRANS,TCCASSUMPTION By IH, then using CCSYM, or CCTRANS CCASSUMPTION.

TCCCONGRUENCE The given derivation looks like

$$\dfrac{\begin{array}{c}\Gamma \vdash A = B : \mathsf{Type} \quad \forall k.\ \Gamma \vDash a_k = b_k \\ |A = B| = |\{a_1/x_1\} \ldots \{a_j/x_j\}\, c = \{b_1/x_1\} \ldots \{b_j/x_j\}\, c|\end{array}}{\Gamma \vDash A = B}\text{TCCCONGRUENCE}$$

From the IHs we know $\forall k.\ \Gamma \vdash a_k = b_k$, so by applying CCCONGRUENCE j times we get

$$\Gamma \vdash \{a_1/x_1\} \ldots \{a_j/x_j\}\, c = \{b_1/x_1\} \ldots \{b_j/x_j\}\, c$$

Then use CCREFL and CCTRANS to get $\Gamma \vdash a = B$.

TCCINJRNG The IH gives $\Gamma \vdash (A_1 \to B_1) = (A_2 \to B_2)$. Then apply CCINJRNG.

TCCINJDOM,TCCIINJDOM,TCCIINJRNG,TCCINJEQ Similar to previous case.

□

Putting together two lemmas we get this version which is quoted in the paper:

***Corollary* 41** (TCC implies LCC)**.** If $\Gamma \vDash a = b$ then $\mathsf{label}\, \Gamma \vdash^{\mathsf{L}} \mathsf{label}\, a = \mathsf{label}\, b$.

Proof. By lemma 40 we have $\Gamma \vdash a = b$, then by lemma 34 we get $\mathsf{label}\, \Gamma \vdash^{\mathsf{L}} \mathsf{label}\, a = \mathsf{label}\, b$. □

***Lemma* 42** (Untyped CC ignores annotations in Γ)**.** If $\Gamma \vdash a = b$ and $|\Gamma| = |\Gamma'|$ then $\Gamma' \vdash a = b$.

Proof. By induction on $\Gamma \vdash a = b$. All the cases are immediate by IH except CCASSUMPTION, were we are given

$$\frac{x : A \in \Gamma \quad \Gamma \vdash A = (a = b)}{\Gamma \vdash a = b}\textsc{CCAssumption}$$

By the IH we know $\Gamma' \vdash A = (a = b)$. From the assumption $|\Gamma| = |\Gamma'|$ we know that there is some $x : A' \in \Gamma'$ with $|A'| = |A|$. By CCREFL we have $\Gamma' \vdash A' = A$, so by CCTRANS we know $\Gamma \vdash A' = (a = b)$. Then conclude by CCASSUMPTION. □

***Lemma* 43** (Untyped CC ignores annotations)**.** If $\Gamma \vdash a = b$ and $|\Gamma| = |\Gamma'|$ and $|a| = |a'|$ and $|b| = |b'|$, then $\Gamma' \vdash a' = b'$.

Proof. By lemma 42 we know $\Gamma' \vdash a = b$, and by CCREFL we know $\Gamma' \vdash a' = a$ and $\Gamma' \vdash b = b'$. Then conclude by CCTRANS. □

***Lemma* 44** (CC doesn't look at type annotations)**.** Suppose $\Gamma \vDash a = b$, and $|\Gamma'| = |\Gamma|$, $|a'| = |a|$ and $|b'| = |b|$, and $\Gamma' \vdash a' : A'$ and $\Gamma' \vdash b' : B'$. Then $\Gamma' \vDash a' = b'$.

Proof. From $\Gamma \vDash a = b$ by lemma 40, we get $\Gamma \vdash a = b$. By lemma 43 we get $\Gamma' \vdash a' = b'$. Then by theorem 39 we get $\Gamma' \vDash a' = b'$. □

D. The untyped congruence closure algorithm and its correctness

The following section gives a precise mathematical definition of our algorithm to decide the $\Gamma \vdash^{\llcorner} a = b$ relation, and a correctness proof. The algorithm was described informally in Section 7.2.

D.1 Flattening

Developing the main rewriting algorithm is easier if the input problem is in a simple, restricted form. So following Nieuwenhuis and Oliveras [23] we first "flatten" the problem by introducing a fresh name for each subterm that occurs in it. We assume that we have an infinite set of *atomic constants* c_i available. The basic idea is that for any context Γ, we can construct an equivalent context with named subterms, e.g. a given assumption $h : f(g\ a) = b$ can be replaced with the set of assumptions

$$\begin{array}{rll} h_1 : & f & = c_1 \\ h_2 : & g & = c_2 \\ h_3 : & a & = c_3 \\ h_4 : & c_2\ c_3 & = c_4 \\ h_5 : & c_1\ c_4 & = c_5 \\ h_6 : & b & = c_6 \\ h : & c_5 & = c_6 \end{array}$$

In the ZOMBIE implementation, the flattening pass works directly on core language expressions. Constants are just integers, and the output of the flattening pass consists of a list of equations in the following Haskell datatype:

```
data EqConstConst = EqConstConst Constant Constant
data EqBranchConst  = EqBranchConst Label [Constant] Constant
type Equation = Either EqConstConst EqBranchConst
```

In addition, there is a table keeping track of additional information about each constant—in particular, whether that constant represents a type which is inhabited by a variable in the context. This is needed to handle the "assumption up to congruence" rule.

In order to reason about the correctness of this process, we need to formalize the input and output of the flattening. We aim to verify the algorithm, not the implementation, so we abstract away from the exact datastructures and instead represent the flattening stage as a transformation from contexts to context. The input is a context where each member is a labelled term (as defined in section C.1). The output is a context containing all the equations (the hs above), and also variable declarations encoding the information about being inhabited. For simplicitly, whenever a constant is marked as inhabited we assume that there is an inhabitant for both the constant and the expression that it names. (When generating core language proofs all constants are replaced with the original core language expressions they named). For a more complete example, the labelled context

$$\begin{array}{rl} x : & F\ ab \\ y : & F\ a\ b = G \end{array}$$

will be transformed into the flat context

$$\begin{array}{rll} h_1 : & a & = c_1 \\ h_2 : & b & = c_2 \\ h_3 : & F\ c_1\ c_2 & = c_3 \\ h_4 : & G & = c_4 \\ h_5 : & (c_3 = c_4) = & c_5 \\ x_1 : & c_3 & \\ x_2 : & F\ c_1\ c_2 & \\ y_1 : & c_5 & \\ y_2 : & (c_3 = c_4) & \end{array}$$

where the h_i represent the list of equations that the algorithm outputs, and the x_i and y_i represent the information that c_3 and c_5 are inhabited.

The treatment of flattening is a bit more subtle than in previous work about first-order logic. In first-order systems, terms and equations are syntactically distinct categories, and one can maintain the invariant that every non-atomic subterm appearing in the flat context has a name. But in our setting there are two sources of equations in the flat context and only some of them have names; in the above example the equation x from the input context has been given the name c_5, but the flat context also contains the new assumptions h_i, and we do not allocate constants naming them (which would lead to infinite regress).

To be precise, the output of the flattening phase is a *flat context*, in the sense of the following definiton.

***Definition* 45** (Flat term)**.** A term is *flat* if it is either an atom a, or a label application $F\,\overline{a_i}$ such that each a_i is an atom.

***Definition* 46** (Flat term over Γ)**.** Let Γ be a context. We say that a term a is *flat over* Γ if a is either an atom, or it is a label applied to a list of atoms $F\,\overline{a_i}$ which is the left-hand-side of an equation in Γ.

***Definition* 47** (Flat context)**.** A context Γ is flat if each variable binding in it is either:

- $x : a$ where a is a flat term over Γ.
- $x : a = b$ where a and b are atoms.
- $x : F\,\overline{a_i} = b$, where a_i and b are atoms, and satisfying the following property: there exists a variable $y : F\,\overline{a_i} \in \Gamma$ iff there exists a variable $z : b \in \Gamma$.

In the above example, the first bullet point corresponds to the x_i and y_i, and the second two bullet points correspond to the h_i.

Given any context Γ we can create an equivalent flat context Γ' by repeatedly picking a subexpression b which is is not yet a left-hand-side of an equation, picking a fresh name x for it, and replacing b with x throughout the context and goal. This procedure is exactly the same as the one by Nieuwenhuis and Oliveras.

However, the proof of its correctness is slightly more complicated. The following lemmas show that this this operation does not change what equations are provable. But in addition, we sharpen the result slightly to specify what the proofs look like: the new equations (the h_i in the above example) can be used as-is as assumptions, there is no need to for the more general assumption-up-to-CC rule. We need the sharpened result to justify that the flattening algorithm is complete even though it only works on the original input context, and does not go on to recursively flatten the new equations that it introduced.

***Lemma* 48.** For any labelled context Γ, and any labelled terms a and b, we have $\Gamma, h : x = b \vdash^{\mathrm{L}} p : a = \{b/x\}\, a$. Furthermore, every use of h in the evidence term p is of the form $h_{\triangleright\mathsf{refl}}$.

Proof. Induction on the structure of a.

- It is the variable x (a nullary label application). By CCAssumption we have $\Gamma, h : x = b \vdash^{\mathrm{L}} h_{\triangleright\mathsf{refl}} : x = b$, as required.
- It is some other application $F\,\overline{a_i}$. Then $\{b/x\}\,(F\,\overline{a_i}) = F\ \overline{\{b/x\}\,a_i}$. By IH we get $\Gamma, h : x = b \vdash^{\mathrm{L}} p_i : a_i = \{b/x\}\, a_i$ and hence by congruence we have $\Gamma, h : x = b \vdash^{\mathrm{L}} \mathsf{cong}\, F\,\overline{p_i}^{\,i} : F\,\overline{a_i} = \{b/x\}\,(F\,\overline{a_i})$.

□

***Lemma* 49** (Naming subterms)**.** Suppose that x does not occur in b, and h is completely fresh. Then there exists p such that $\{b/x\}\,\Gamma \vdash^{\mathrm{L}} p : \{b/x\}\, a_1 = \{b/x\}\, a_2$ iff there exists p' such that $\Gamma, h : x = b \vdash^{\mathrm{L}} p' : a_1 = a_2$. Futhermore, any use of h in p' is of the form $h_{\triangleright\mathsf{refl}}$.

Proof. We prove the two directions by separate inductions. For the the "⇒" direction, the cases are:

CCRefl We know that $\{b/x\}\, a_1 \equiv \{b/x\}\, a_2$. Apply lemma 48 to get $\Gamma, h : x = b \vdash^{\mathrm{L}} p_1 : a_1 = \{b/x\}\, a_1$ and $\Gamma, h : x = b \vdash^{\mathrm{L}} p_2 : \{b/x\}\, a_2 = a_2$, then conclude by transitivity.

CCSym, CCTrans Directly by IH.

CCAssumption We are given the derivation

$$\frac{y : \{b/x\}\, A \in \{b/x\}\,\Gamma \qquad \{b/x\}\,\Gamma \vdash^{\mathrm{L}} q : \{b/x\}\, A = (\{b/x\}\, a_1 = \{b/x\}\, a_2)}{\{b/x\}\,\Gamma \vdash^{\mathrm{L}} y_{\triangleright q} : \{b/x\}\, a_1 = \{b/x\}\, a_2}$$

(Where $y : A \in \Gamma$). By the IH, we have $\Gamma, h : x = b \vdash^{\mathrm{L}} q' : A = (a_1 = a_2)$. Then apply CCAssumption again.

CCCong We are given derivation $\{b/x\}\,\Gamma \vdash^{\mathrm{L}} \mathsf{cong}\, F\,\overline{p_k}^{\,k} : \{b/x\}\,(F\,\overline{a_i}) = \{b/x\}\,(F\,\overline{b_i})$. Note that $\{b/x\}\,(F\,\overline{a_i}) \equiv F\overline{\{b/x\}a_i}$, then apply the IHs for the p_k.

CCInj Similar to the previous case.

The cases for the "⇐" direction are:

CCRefl Directly by CCRefl.

CCSymm, CCTrans Immediate from IH.

CCPASSUMPTION We are given the the derivation

$$\frac{y : A \in (\Gamma, h : x = b) \qquad \Gamma, h : x = b \vdash^{\mathsf{L}} q : A = (a_1 = a_2)}{\Gamma, h : x = b \vdash^{\mathsf{L}} y_{\triangleright q} : a_1 = a_2}$$

There are two cases. If $x \equiv h$, we know $\Gamma, h : x = b \vdash^{\mathsf{L}} q : (x = b) = (a_1 = a_2)$. By the IH we have $\{b/x\}\,\Gamma \vdash^{\mathsf{L}} q' : (b = b) = (\{b/x\}\,a_1 = \{b/x\}\,a_2)$ By CCPINJ we get $\{b/x\}\,\Gamma \vdash^{\mathsf{L}} q_1 : b = \{b/x\}\,a_1$ and $\{b/x\}\,\Gamma \vdash^{\mathsf{L}} q_2 : b = \{b/x\}\,a_2$. Then conclude by symmetry and transitivity.

Otherwise, $y : A \in \Gamma$. By the IH we have $\{b/x\}\,\Gamma \vdash^{\mathsf{L}} q' : \{b/x\}\,A = (\{b/x\}\,a_1 = \{b/x\}\,a_2)$, so $\{b/x\}\,\Gamma \vdash^{\mathsf{L}} y_{\triangleright q'} : \{b/x\}\,a_1 = \{b/x\}\,a_2$, as required.

CCPCONG,CCPINJ From IH, using the fact that $\{b/x\}\,(F\,\overline{a_i}) \equiv F\overline{\{b/x\}a_i}$.

In the assumption case for the, we are given that $a_1 = a_2 \in (\Gamma, h : x = b)$. If the equation used was h itself we must prove $\{b/x\}\,x = \{b/x\}\,b$ which is certainly true.

Otherwise, we have $(a_1 = a_2) \in \Gamma$, and we must prove $\{b/x\}\,\Gamma \vdash \{b/x\}\,a_1 = \{b/x\}\,a_2$; this follows directly by the assumption rule. ☐

***Lemma* 50** (Redundant equal assumptions)**.** If $\Gamma \vdash^{\mathsf{L}} b_1 = b_2$, then $\Gamma, x_1 : b_1 \vdash a_1 = b_2$ iff $\Gamma, x_1 : b_1, x_2 : b_2 \vdash a_1 = a_2$

Proof. The "$\Leftarrow$" direction is a trivial induction. The "$\Rightarrow$" direction is by induction on $\Gamma, x_1 : b_1 \vdash a_1 = b_2$. The only interesting case is the assumption case, in the case when x_2 is the used assumption. Then we are given the derivation

$$\frac{\Gamma, x_1 : b_1, x_2 : b_2 \vdash b_2 = (a_1 = a_2)}{\Gamma, x_1 : b_1, x_2 : b_2 \vdash a_1 = a_2}$$

By IH we get $\Gamma, x_1 : b_1 \vdash b_2 = (a_1 = a_2)$. Then by transitivity we have $\Gamma, x_1 : b_1 \vdash b_1 = (a_1 = a_2)$, and conclude by using assumption x_1. ☐

***Lemma* 51** (Flattening contexts)**.** For any triple (Γ, a_1, a_2), we can find a triple (Γ', a_1', a_2'), where Γ' contains two sets of assumptions x_i and h_i, which satisfies the following:

1. For all $x_i : A \in \Gamma'$, the expression A is a flat term over Γ'.
2. For all $h_i : A \in \Gamma'$, A is an equation of the form mentioned in one of the second two bullet points of definition 47.
3. There exists some p such that $\Gamma \vdash^{\mathsf{L}} p : a_1 = a_2$ if and only if there exists some p' such that $\Gamma' \vdash^{\mathsf{L}} p' : a_1' = a_2'$. Furthermore, every use in p' of an assumption from the set h_i has the form $h_{i \triangleright \mathsf{refl}}$ (i.e. the conversion is just refl).

In particular, (1) and (2) implies that Γ' is a flat context.

Proof. We begin with the context Γ, and let the assumptions in it be the original set of assumptions x. Then we repeatedly use lemma 49 to add additional equations h until properties (1) is satisfied, while maintaining (2) and (3) as invariants. We write $\Gamma_0, \Gamma_1, \ldots$ for the intermediate contexts.

The original context $\Gamma_0 \equiv \Gamma$ trivially satisfies (2) and (3), since the set of assumptions h_i is empty.

Now let Γ_k be some intermediate context. If all the assumptions $x_i : A \in \Gamma_k$ are already over flat terms over Γ_k then we are done. Otherwise, A is a labelled term, so we pick a subterm of it of the form $b \equiv F\,\overline{a_i}$ with a_i atomic, pick a fresh atom c, and replace all occurances of b with c everywhere in Γ_k, a_1 and a_2. Call the resulting context Γ_k', so that $\Gamma_k \equiv \{b/c\}\Gamma_k'$. The next context is then $\Gamma_{k+1} \equiv \Gamma_k', h : b = c, x' : b$ if b occured as an assumption in Γ_k, and b$\Gamma_{k+1} \equiv \Gamma_k', h : b = c$ otherwise. We check that Γ_{k+1} still satisfies the invariants. For (1), x' is indeed a flat term over the context (thanks to h). For (2), the new equation is of the application-constant form, and either neither side is inhabited, or x and x' inhabit the two sides.

For (3), we consider the case where $\Gamma_{k+1} \equiv \Gamma_k', h : b = c, x' : b$ (the case when there is no assumption x' is simpler). We need to show

There exists some p such that $\Gamma_k \vdash^{\mathsf{L}} p : a_1 = a_2$ if and only if there exists some p' such that $\Gamma_k', h : b = c, x' : b \vdash^{\mathsf{L}} p' : a_1' = a_2'$ (with uses of hs restricted).

Lemma 49 gives us that $\Gamma_k \vdash^{\mathsf{L}} p : a_1 = a_2$ iff $\Gamma_k', h : b = c \vdash^{\mathsf{L}} p' : a_1' = a_2'$. And lemma 50 gives $\Gamma_k', h : b = c \vdash^{\mathsf{L}} p' : a_1' = a_2'$ iff $\Gamma_k', h : b = c, x' : b \vdash^{\mathsf{L}} p' : a_1' = a_2'$, because $x : c \in \Gamma_k'$. ☐

D.2 Main Algorithm

The state of the algorithm consists of:

- A list E of *pending equations* to be processed.
- A *representatives* table, which maps each constant c to its Union-Find representative $c' = r(c)$. Along which each representative, we store information about that equivalence class:
  - The *equality list*, $Q(c)$. The set of pairs of constants (a, b) such that $a = b$ is in this equivalence class of c'.

- The *injectivity list*, $I(c)$. The set of tuples $(Ax_1, \ldots x_n)$ such that A is injective and $A\ x_1 \ldots x_n$ is in the equivalence class of c'.
- The *use list*, $U(c)$: the set of input equations $y = A\ x_1 \ldots x_n$ such that c' is the representative of one of the x_i.
- The *assumption flag*, $A(c)$. A Boolean tracking any member of the equivalence class that was inhabited by a variable in the context.

We will overload notation slightly to let $Q(a)$ mean $Q(r(a))$ when a is not the representative of its class, and similar for I, U, and A.

- The *lookup table* (a.k.a signature table), S: maps tuples $(A, x_1, \ldots x_n)$ to an input equation $y = A\ x_1 \ldots x_n$, if such an equation exists, or to the undefined value $\perp$ otherwise.

Of these, $I(c)$, $Q(c)$, and $A(c)$ are additions which were not in the Nieuwenhuis-Oliveras algorithm.

The algorithm is initialized as follows:

$$
\begin{array}{lll}
E_0 & = & \text{All the given equations in } \Gamma \\
r_0(c) & = c & \text{for all constants } c \text{ in the problem} \\
Q_0(c) & = \emptyset & \text{for all constants } c \\
I_0(c) & = \emptyset & \text{for all constants } c \\
U_0(c) & = \emptyset & \text{for all constants } c \\
A_0(c) & = \mathsf{true} \text{ iff } x : c \in \Gamma & \\
S_0(F, a_1, \ldots, a_n) & = \perp & \text{for all labels and constants}
\end{array}
$$

The algorithm then proceeds by considering the pending equations one by one, updating the state and sometimes adding additional pending equations. We can show it symbolically as a transition system between tuples containing the state. (In the "merge" rule, we show the case where a rather than b is picked as the representative by the union operation, but this choice does not affect correctness, and in practice the implementation will choose one or the other depending on the size of the equivalence classes).

$$
\begin{array}{rl}
\textsc{Trivial} & (E \cup \{a = b\}, r, Q, I, U, A, S) \\
\Longrightarrow & (E, r, Q, I, U, A, S) \\
& \text{when } r(a) = r(b) \text{ already}
\end{array}
$$

$$
\begin{array}{rl}
\textsc{Merge} & (E \cup \{a = b\}, r, Q, I, U, A, S) \\
\Longrightarrow & (E \cup \{a_i = b_i \mid (F\ a_1 \ldots a_n) \in I(a) \text{ and } (F\ b_1 \ldots b_n) \in I(b)\} \\
& \quad \cup U(b) \\
& \quad \cup \{c = c' \mid (c, c') \in Q(a) \wedge A(b) \wedge \neg A(a))\} \\
& \quad \cup \{c = c' \mid (c, c') \in Q(b) \wedge A(a) \wedge \neg A(b))\}, \\
& r', Q', I', U, A', S) \\
& \text{where } r'(b) = r(a),\ Q'(a) = Q(a) \cup Q(b),\ I'(a) = I(a) \cup I(b), \text{ and } A'(a) = A(a) \vee A(b)
\end{array}
$$

$$
\begin{array}{rl}
\textsc{Update1} & E \cup \{F\ a_1 \ldots a_n = a\}, r, Q, I, U, A, S) \\
\Longrightarrow & (E', r, Q', I', U', A, S') \\
& \text{where } S'(F, a_1, \ldots, a_n) = (F\ a_1 \ldots a_n = a) \\
& \text{when } S(F, a_1, \ldots, a_n) = \perp
\end{array}
$$

$$
\begin{array}{rl}
\textsc{Update2} & (E \cup \{F\ a_1 \ldots a_n = a\}, r, Q, I, U, A, S) \\
\Longrightarrow & (\quad E' \cup \{a = b\}, r, Q', I', U', A, S) \\
& \text{when } S(F, a_1, \ldots, a_n) = (F\ b_1 \ldots b_n = b)
\end{array}
$$

Where in the Update1 and Update2 rules,

$$
\begin{array}{ll}
E' & = E \cup \{a_i = b_i \mid (F\ b_1 \ldots b_n) \in I(a)\} \cup \{c = c' \mid \text{if } F\ a_1 \ldots a_n \text{ is } c = c' \text{ and } A(a)\ \} \\
Q'(a) & = Q(a) \cup \{c = c' \mid \text{if } F\ a_1 \ldots a_n \text{ is } c = c'\} \\
I'(a) & = I(a) \cup \{F\ a_1 \ldots a_n \mid \text{if } F \text{ is injective}\} \\
U'(a_i) & = U(a_i) \cup (F\ a_1 \ldots a_n = a) \quad \text{for } 1 \le i \le n
\end{array}
$$

D.3 Soundness

***Lemma* 52** (Invariants for soundness)**.** Suppose $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$ is the initial state correponding to a flat context Γ, and $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0) \Longrightarrow^* (E, r, Q, I, U, A, S)$. Then

1. If $(a = b) \in E$, then $\Gamma \vdash^{\mathsf{L}} a = b$.
2. If $r(a) = b$, then $\Gamma \vdash^{\mathsf{L}} a = b$.
3. If $(a = b) \in Q(c)$, then $\Gamma \vdash^{\mathsf{L}} c = (a = b)$.
4. If $F\ \overline{a_i} \in I(c)$, then $\Gamma \vdash^{\mathsf{L}} c = (F\ \overline{a_i})$ and F is injective.
5. If $U(c) = (F\ \overline{a_i} = a)$, then $\Gamma \vdash^{\mathsf{L}} F\ \overline{a_i} = a$.

$$\frac{}{a \approx_E a}\textsc{EQRefl} \qquad \frac{\begin{array}{c} a \approx_E b \\ b \approx_E c \end{array}}{a \approx_E c}\textsc{EQTrans} \qquad \frac{a = b \in E}{a \approx_E b}\textsc{EQAssumption} \qquad \frac{a \approx_E b}{b \approx_E a}\textsc{EQSymm}$$

Figure 17. The equivalence relation generated by a set of equations E

6. If $S(F, a_1, \ldots, a_n) = (F\,\overline{b_i} = b)$, then $\Gamma \vdash^{\mathsf{L}} F\,\overline{b_i} = b$ and $\Gamma \vdash^{\mathsf{L}} F\,\overline{a_i} = b$.
7. If $A(c) = \mathsf{true}$, then there exists some $x : A \in \Gamma$ such that $\Gamma \vdash^{\mathsf{L}} c = A$.

Proof. We first check that all then invariants hold in the initial state $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$. (1) is true because each equation in E_0 is a hypothesis from Γ. (2) is true because r_0 is just the reflective relation. (3–6) are vacuously true since the sets Q,I,U and S are all empty. And (7) holds because of how A was initialized.

Next, we check that all the transitions of the algorithm preserves the invariants. In the TRIVIAL transition the only component of the state that changes is E, and $E' \subset E$ gets smaller so the invariant is trivially preserved. MERGE and UPDATE1/2 add additional equations, but it is easy to see that they are justified by the $\Gamma \vdash a = b$ relation.

(In the implementation, the datastructures for E,r and S store not only the terms a and b, but also proof terms $\Gamma \vdash^{\mathsf{L}} p : a = b$. Each transition constructs new proof terms from the old). □

D.4 Completeness

The completeness proof follows the same strategy as the proof by Corbineau [11]. We prove that at the end of a run of the algorithm, the union-find structure r has enough links to validate all the proof rules of the $\Gamma \vdash^{\mathsf{L}} a = b$ relation—in particular the assumption, congruence, and injectivity rules.

The invariant properties of the relation are stated in terms of the equivalence relations generated by a sets of equations. We let the letters E and R range over lists of equations,

$$E, R \quad ::= \quad \cdot \mid E, a = b$$

and write $a \approx_E b$ for the equivalence relation generated by such a list. In other words, the relation defined by the rules in Figure 17.

The equivalence relations satisfies some simple properties:

***Lemma* 53.** If $b \approx_{(E,a=a')} b'$, then either $b \approx_E b'$, or $b \approx_E a$ and $a' \approx_E b$, or $b \approx_E a'$ and $a \approx_E b$.

Proof. Induction on the judgement $b \approx_{(E,a=a')} b'$. □

***Lemma* 54.** If $a \approx_E a'$, then $b \approx_{(E,a=a')} b'$ iff $b \approx_E b'$.

Proof. The "$\Leftarrow$" direction is an easy induction. For the "$\Rightarrow$" direction, by lemma 53 either we have $b \approx_E b'$ (and we are done), or else the equation was used. If the equation was used we have either $b \approx_E a$ and $a' \approx_E b$, or $b \approx_E a'$ and $a \approx_E b$. Either way, the conclusion follows by transitivity and symmetry. □

***Lemma* 55.** If $a \approx_E b$ or $b \approx_E a$, and the is is not an instance of reflexivity (i.e. $a \not\equiv b$), then E contains some equation of the form $a = c$ or $c = a$.

Proof. Easy induction. □

In a given a state (E, r, Q, I, U, A, S) of the algorithm, we write E for the set of equations occuring in the first component, and we write R to denote the content of r and S interpreted as a a set of equations according to the following scheme:

- One equation equation $c = c'$ whenever $r(c) = c'$.
- One equation equation $F\,\overline{a_i} = b$ whenever $\forall k. r(a_k) = a'_k$ and $S(F, a'_1, \ldots, a'_n) = (F\,\overline{b_i} = b)$.

Note that R is finite, because both r and S have finite domains. We use the notation $E \setminus E'$ to denote set-difference.

In all the following we assume that the list E has no duplicates, so we can equivocate between treating it as a set and as a list. This makes it easier to state the invariants of the algorithm (in particular invariant 2 below). In practice, if the list *does* contain duplicates they will eventually be discarded by the rule TRIVIAL, so when implementing the algorithm there is no need to preprocess the list to remove them.

***Lemma* 56** (Monotonicity of $\approx_{E,R}$)**.** If $(E, r, Q, I, U, A, S) \Longrightarrow (E', r', Q', I', U', A', S')$ and $c_1 \approx_{E,R} c_2$, then $c_1 \approx_{E',R'} c_2$.

Proof. We consider each of the transitions in turn.

Trivial We already had the equation $a = b \in R$, so $E \cup R \equiv E' \cup R'$.

Merge We deleted the equation $a = b$ from E, and added the equation $r(a) = r(b)$ to R. By transitivity we can derive $a \approx r(a) \approx r(b) \approx b$. Then appeal to lemma 54.

Update1 We deleted the equation $F\,\overline{a_i} = a$ from E and added it to R, so $E \cup R \equiv E' \cup R'$.

Update2 By the definition of R, we already had $F\,\overline{a_i} = b \in R$. Now we deleted $F\,\overline{a_i} = a$ from E, and instead added $a = b$. By transitivity we can derive $F\,\overline{a_i} \approx b \approx a$. Then appeal to lemma 54.

□

We can now state the invariants of the algorithm.

***Lemma* 57** (Invariants for completeness of CC algorithm)**.** Suppose $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$ is the initial state correponding to a flat context Γ, and $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0) \Longrightarrow^* (E, r, Q, I, U, A, S)$. Then

1. If $x : A \in \Gamma$ then for all a, b, if $A \approx_R (a = b)$ then $a \approx_{E,R} b$.
2. If for all $0 \leq i < n$ we have $a_i \approx_R b_i$, and both $F\,\overline{a_i}$ and $F\,\overline{b_i}$ are left-hand-sides of equations in $E_0 \setminus E$, then $F\,\overline{a_i} \approx_{E,R} F\,\overline{b_i}$.
3. For all $\overline{a_i}$ and $\overline{b_i}$, if $F\,\overline{a_i} \approx_R F\,\overline{b_i}$ and F is injective, then $\forall k. a_k \approx_{E,R} b_k$.
4. If $F\,\overline{a_i} = b \in (E_0 \setminus E)$, then for all $0 \leq i < n$ we have $(F\,\overline{a_i} = b) \in U(a_i)$.
5. If $F\,\overline{a_i} = a \in (E_0 \setminus E)$ and $r(a_k) = a'_k$, then $S(F\,\overline{a_i}') = (F\,\overline{b_i} = b)$ for some equation such that $b \approx_{E,R} a$ and $b_k \approx_R a_k$. And conversely, if $S(F\,\overline{a_i}') = (F\,\overline{b_i} = b)$, then the equation $F\,\overline{a_i} = a \in (E_0 \setminus E)$ and $b \approx_{E,R} a$ and $b_k \approx_R a_k$.
6. If $c \approx_R (a = b)$, then $(a' = b') \in Q(c)$, for some constants a' and b' such that $a \approx_R a'$ and $b \approx_R b'$.
7. if $c \approx_R F\,\overline{a_i}$ for some injective label F, then $F\,\overline{a_i} \in I(c)$.
8. $A(c)$ iff $c \approx_R A$ for some A such that $x : A \in \Gamma$.
9. All equations in E, S and U are between flat terms. Also, if an equation has the form the form $F\,\overline{a_i} = a$ (label application vs atomic constant), then that equation was present in E_0, and there exists a variable $x : F\,\overline{a_i} \in \Gamma$ iff there exists a variable $y : a \in \Gamma$

Proof. We first must check that these invariants hold for the initial state $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$.

1. In the initial state R is just the reflexive relation, so the statement simplifies to "if $x : a = b \in \Gamma$ then $a \approx_{E,R} b$". In the initial state corresponding to Γ, we have $(a = b) \in E_0$, so this is true.
2. $E_0 \setminus E$ is empty, so vacuously true.
3. R is the reflexive relation, so the only case we worry about is a reflexive equation $F\,\overline{a_i} \approx_R F\,\overline{a_i}$. Then we certainly also have $a_k \approx_{E,R} a_k$.
4. $E_0 \setminus E$ is empty, so vacuously true.
5. Both $E_0 \setminus E$ and S are empty, so both directions are vacuously true.
6. R is the reflexive relation, so we can never have an atom $\approx$ a label application.
7. Similar to invariant 6.
8. R is the reflexive relation, so $A(c)$ should be inhabited if the constant c itself is inhabited by a variable. This is exactly how A is initialized.
9. S_0 and U_0 are empty, so we only need to consider the equations in E_0. For these, the invariant is just restating part of the assumption that Γ is a flat context (definition 47).

Next, we check that the invariants are preserved by each transition $(E, r, Q, I, U, A, S) \Longrightarrow (E', r', Q', I', U', A', S')$. The cases are:

TRIVIAL Here $E = E', a = b$ and $R = R'$. By the precondition to the rule we know $a \approx_R b$, so by lemma 54 the relations $\approx_{E,R}$ and $\approx_{E',R'}$ coincide. And since $R = R'$ the relations $\approx_R$ and $\approx_{R'}$ coincide trivially. Finally, the set of expressions $F\,\overline{a_i}$ which appear as left-hand sides in $E_0 \setminus E$ and $E_0 \setminus E'$ are the same (since the only equation that changed was an atom-atom equation). It is then easy to see that all the invariants are preserved.

MERGE In this transition, S is unchanged and we added one link to r. So $R' = (R, b = a')$, where we write $a' = r(a)$.

1. We are given some A, c_1, c_2 such that $A \approx_{R'} (c_1 = c_2)$, and we must show $c_1 \approx_{E',R'} c_2$.

   By lemma 53, there are two cases. Either the new equation was not used, i.e. $A \approx_R (c_1 = c_2)$. Then by the IH for the previous step we have $c_1 \approx_{E,R} c_2$. By monotonicity (lemma 56) $c_1 \approx_{E',R'} c_2$ as required.

   Otherwise the new equation was used, so we have $A \approx_R b$ and $a' \approx_R (c_1 = c_2)$ (or the symmetric $A \approx_R a'$ and $b \approx_R (c_1 = c_2)$; we show the first case w.l.o.g.). By invariant 8 we know that $A(b) = \texttt{true}$, and by invariant 6 we know that $(c'_1 = c'_2) \in Q(a')$ for $c'_1 \approx_R c_1$ and $c'_2 \approx_R c_2$.

Now proceed by cases on the value of $A(a')$. If $A(a') = \mathtt{true}$, then by invariant 8 we know that there is some $y : A' \in \Gamma$ such that $A' \approx_R a'$. So by invariant 1 we have $c_1 \approx_{E,R} c_2$. By monotonicity (lemma 56) $c_1 \approx_{E',R'} c_2$ as required.

Otherwise, $A(a') = \mathsf{false}$. We have $A \approx_R b$, so by invariant 8 we known $A(b) = \mathtt{true}$. In other words, we have $A(b) \wedge \neg A(a)$. So the transition rule MERGE will add the equation $c_1' = c_2'$ to E'. Then $c_1 \approx_{E',R'} c_2$ using that new equation.

2. We are given some $F\,\overline{c_i}$ and $F\,\overline{c_i}' \in E_0 \setminus E'$, such that $\forall i. c_i \approx_{R'} c_i'$, and we need to show $F\,\overline{c_i} \approx_{E',R'} F\,\overline{c_i}'$.

   Apply lemma 53 to each of the $c_i \approx_{R'} c_i'$. Suppose that all of them fall in the first the first case, so the new equation was not used and we have $c_i \approx_R c_i'$. Then by invariant 2 we have $F\,\overline{c_i} \approx_{E,R} F\,\overline{c_i}'$. By monotonicity (lemma 56) $F\,\overline{c_i} \approx_{E',R'} F\,\overline{c_i}'$ as required.

   Otherwise, there is at least one k such that the new equation $b = a'$ was used. That is, we have $c_k \approx_R b$ and $a' \approx_R c_k'$ (or the symmetric case $c_k \approx_R a'$ and $b \approx_R c_k'$; we show the first case w.l.o.g.). So in particular c_k and b have the same representative. Now $E_0 \setminus E \supseteq E_0 \setminus E'$, so $F\,\overline{c_i} \in E_0 \setminus E$. Then by invariant 4 we have $(F\,\overline{c_i} = c_0) \in U(b)$. So by the transition rule MERGE we have $(F\,\overline{c_i} = c_0) \in E'$, contradicting the assumption that $F\,\overline{c_i} \in E_0 \setminus E'$.

3. We are given $F\,\overline{c_i} \approx_{R'} F\,\overline{c_i}'$ and must show $c_k \approx_{E',R'} c_k'$. By lemma 53 we must consider two cases.

   Either $F\,\overline{c_i} \approx_R F\,\overline{c_i}'$. Then by invariant 3 we have $F\,\overline{c_i} \approx_{E,R} F\,\overline{c_i}'$, and by monotonicity (lemma 56) $F\,\overline{c_i} \approx_{E',R'} F\,\overline{c_i}'$ as required.

   Otherwise we have $F\,\overline{c_i} \approx_R b$ and $a' \approx_R F\,\overline{c_i}'$ (or the symmetric case). So by invariant 7 we have $F\,\overline{c_i} \in I(b)$ and $F\,\overline{c_i}' \in I(a)$. So by the transition rule MERGE the equation $c_k = c_k'$ is explicitly added to E', and we have $c_k \approx_{E',R'} c_k'$ as required.

4. $F\,\overline{a_i} = a \in (E_0 \setminus E')$. The only equation which changed was an atom-atom equation, so we also have $F\,\overline{a_i} = a \in (E_0 \setminus E)$. Then appeal to invariant 4 for the previous state.

5. For the first direction, suppose $F\,\overline{a_i} = a \in (E_0 \setminus E')$. The only equation which changed was an atom-atom equation, so we also have $F\,\overline{a_i} = a \in (E_0 \setminus E)$. Then by invariant 5 for the previous state, we have $S(F\,\overline{a_i}) = (F\,\overline{b_i} = b)$ which are suitably $\approx_{E,R}$. By monotonicity they are still $\approx_{E',R}$.

   For the converse direction, suppose that $(F\,\overline{b_i} = b)$ is in the range of S. Since the transition rule did not change S, it must still be in the range of S in the previous state. So by the invariant $F\,\overline{b_i} = b \in (E_0 \setminus E)$, and the subterms are suitably $\approx_{E,R}$. Similar to the previous paragraph, it must also be in $(E_0 \setminus E')$, and by monotonicity the subterms are still $\approx_{E',R'}$.

6. We are given some atoms c, c_1, c_2 such that $c \approx_{R'} (c_1 = c_2)$, and we must show $c_1' = c_2' \in Q'(c)$.

   By lemma 53, there are two cases. Either the new equation was not used, i.e. $c \approx_R (c_1 = c_2)$. Then by the IH for the previous step we have $(c_1' = c_2') \in Q(c)$ and hence in $Q'(c)$.

   Otherwise the new equation was used, so we have $c \approx_R b$ and $a' \approx_R (c_1 = c_2)$ (or the symmetric $A \approx_R a'$ and $b \approx_R (c_1 = c_2)$; we show the first case w.l.o.g.). By invariant 6 we have $(c1' = c2') \in Q(a)$, and hence in $Q'(c) \equiv Q'(a') \equiv Q(a) \cup Q(b)$.

7. Similar to invariant 6.

8. Similar to invariant 6.

9. The transition leaves S and U unchanged. The equations added to E are either atom-atom, or they came from U and therefore have the required form by invariant 9 for the previous state.

UPDATE1 In this case

$$\begin{array}{lcl} E' & = & (E \setminus (F\,\overline{a_i} = a)) \cup \{c = c' \mid \text{if } F\ a_1 \ldots a_n \text{ is } c = c' \text{ and } A(a)\ \} \\ R' & = & R, F\,\overline{a_i} = a \end{array}$$

1. We are given some $A, c1, c2$ such that $A \approx_{R'} (c_1 = c_2)$, and we must show $c_1 \approx_{E',R'} c_2$.

   By lemma 53, there are two cases. Either the new equation was not used, i.e. $A \approx_R (c_1 = c_2)$. Then by invariant 1 we have $c_1 \approx_{E,R} c_2$. By monotonicity (lemma 56) $c_1 \approx_{E',R'} c_2$ as required.

   Otherwise the new equation was used, which can happen in two ways.

   - We have $A \approx_R F\,\overline{a_i}$ and $a \approx_R (c_1 = c_2)$.

     By lemma 55, unless $A \equiv F\,\overline{a_i}$ that means that R must contain some equation mentioning $F\,\overline{a_i}$. However, this is impossible: each equation in R comes either from r (but this only relates constants, not label applications) or from S (but we know as a premise to the rule that $S(F\,\overline{a_i}) = \bot$).

     On the other hand, if $A \equiv F\,\overline{a_i}$, then the assumption says that $x : (F\,\overline{a_i}) \in \Gamma$, so by invariant 9 we know that $x : a \in \Gamma$. So by invariant 1 we know $c_1 \approx_{E,R} c_2$, and hence by monotonicity $c_1 \approx_{E',R'} c_2$.

   - We have $A \approx_R a$ and $F\,\overline{a_i} \approx_R (c_1 = c_2)$.

     By invariant 8 we then have $A(a) = \mathtt{true}$. So by the transition rule UPDATE1 the equation $c_1 = c_2$ is explicitly added to E', and we have $c_1 \approx_{E',R'} c_2$ as required.

2. We are given some $G\,\overline{c_i}$ and $G\,\overline{c_i}' \in E_0 \setminus E'$, such that $\forall i. c_i \approx_{R'} c_i'$, and we need to show $G\,\overline{c_i} \approx_{E',R'} G\,\overline{c_i}'$.

   Apply lemma 53 to all the $c_i \approx_{R'} c_i'$. If the new equation was not used for any of them, we have $\forall i. c_i \approx_R c_i'$. Using the assumption $G\,\overline{c_i} \in E_0 \setminus E'$, invariant 5, and the fact that $S(F\,\overline{a_i}) = \bot$ we know that $G\,\overline{c_i} \not\equiv F\,\overline{a_i}$. This means that we must also have $G\,\overline{c_i} \in E_0 \setminus E$, and similar for $G\,\overline{c_i}'$. Hence by invariant 2 for the previous state and monontonicity we get $G\,\overline{c_i} \approx_{E',R'} G\,\overline{c_i}'$.

   Otherwise, the new equation $F\,\overline{a_i} = a$ was used for at least one c_k, which can happen in two ways.

   - We have $c_k \approx_R F\,\overline{a_i}$ and $a \approx_R c_k'$.

     By lemma 55, unless $A \equiv F\,\overline{a_i}$ that means that R must contain some equation mentioning $F\,\overline{a_i}$. However, this is impossible: each equation in R comes either from r (but this only relates constants, not label applications) or from S (but we know as a premise to the rule that $S(F\,\overline{a_i}) = \bot$).

     So we must have $c_k \equiv (F\,\overline{a_i})$. That means that $G\,\overline{c_i}$ has the form $G\ c_1 \ldots (F\,\overline{a_i}) \ldots c_n$. However, according to invariant 9, $G\,\overline{c_i}$ should be a flat term, so this also cannot happen.

   - We have $c_k \approx_R a$ and $F\,\overline{a_i} \approx_R c_k'$.

     The reasoning in this case is similar, using c_k' instead of c_k.

3. We are given some injective G such that $G\,\overline{c_i} \approx_{R'} G\,\overline{c_i}'$, and we must show $c_k \approx_{E',R'} c_k'$.

   By lemma 53, the new equation is either used or not. If not, we have $G\,\overline{c_i} \approx_R G\,\overline{c_i}'$, so by invariant 3 we get $c_k \approx_{E,R} c_k'$ and hence by monotonicity (lemma 56) $c_k \approx_{E',R'} c_k'$ as required.

   Otherwise the equation is used and we have either $G\,\overline{c_i} \approx_R F\,\overline{a_i}$ and $a \approx_R G\,\overline{c_i}'$, or the symmetric situation. W.l.o.g. we consider the first case.

   By lemma 55, unless $G\,\overline{c_i} \equiv F\,\overline{a_i}$, there must be some equation in R involving $F\,\overline{a_i}$. But that is impossible by invariant 5, since by the premise to the rule UPDATE1 we know that $S(F\,\overline{a_i}) = \bot$.

   On the other hand, if $G\,\overline{c_i} \equiv F\,\overline{a_i}$, then we are given a new equation $F\,\overline{c_i} = a$ and we know $a \approx_R F\,\overline{c_i}'$. So by invariant 7 we know $F\,\overline{c_i} \in I(a)$. So the transition rule UPDATE1 adds the equation $ck = ck'$ to E', and we have $c_k \approx_{E',R'} c_k'$.

4. Supposed $(G\,\overline{c_i} = c) \in (E_0 \setminus E')$. The set $E_0 \setminus E$ contains all equations in $E_0 \setminus E'$ except for $F\,\overline{a_i} = a$. So there are two cases. If $(G\,\overline{c_i} = c) \not\equiv (F\,\overline{a_i} = a)$, then we also have $(G\,\overline{c_i} = c) \in (E_0 \setminus E)$, and can appeal to invariant 4 for the previous state. MERGE transition. Otherwise, if $(G\,\overline{c_i} = c) \equiv (F\,\overline{a_i} = a)$, then the transition rule explicitly adds the equation to U'.

5. For the first direction, suppose $G\,\overline{c_i} = c \in (E_0 \setminus E')$. The set $E_0 \setminus E$ contains all equations in $E_0 \setminus E'$ except for $F\,\overline{a_i} = a$. So there are two cases. If $(G\,\overline{c_i} = c) \not\equiv (F\,\overline{a_i} = a)$, then we also have $(G\,\overline{c_i} = c) \in (E_0 \setminus E)$. So we can use similar reasoning as in the corresponding case for the MERGE transition. Otherwise, if $(G\,\overline{c_i} = c) \equiv (F\,\overline{a_i} = a)$, then in the new state we have $S'(F, a_1, \ldots, a_n) = (F\,\overline{a_i} = a)$. Certainly $a_k \approx_{R'} a_k$ and $a \approx_{E',R'} a$, as required.

   For the converse direction, suppose that $(G\,\overline{c_i} = c)$ is in the range of S'. Again there are two cases. If is was already in the range of S, we reason similarly to the corresponding case for the MERGE transition. Otherwise, if it is the new equation, then by invariant 9 that equation is in E_0, and by the transition rule it is no longer in E', so it is in $(E_0 \setminus E')$ as required.

6. We are given some $c \approx_{R'} (c_1 = c_2)$, and must show that some suitable $(c_1' = c_2') \in Q'(c)$.

   By lemma 53, the new equation from S' is either used or not. If not, we have $c \approx_R (c_1 = c_2)$, and get $(c_1 = c_2) \in Q(c)$ by invariant 6 for the previous state. Otherwise the equation was used, which can happen in two ways:

   - $c \approx_R F\,\overline{a_i}$ and $a \approx_R (c_1 = c_2)$. But we know that c and $F\,\overline{a_i}$ are different (one is an atom and one is a label application), so by lemma 55 that would mean that R contains an equation mentioning $F\,\overline{a_i}$, which is impossible since $S(F\,\overline{a_i} = \bot$.

   - $c \approx_R a$ and $F\,\overline{a_i} \approx_R (c_1 = c_2)$. By reasoning similar to the previous paragraph this can only happen if $F\,\overline{a_i} \equiv (c_1 = c_2)$. In that case we have $(c_1 = c_2) \in Q'(c) \equiv Q'(a)$, since it was explicitly added by the transition rule UPDATE1.

7. Similar to invariant 6.

8. Similar to invariant 6.

9. We modify S and U by adding the equation $F\,\overline{a_i} = a$; this equation comes from E so by the invariant from the previous state it is good. And all the new equations in E' are atom-atom.

UPDATE2 In this transition $R' = R$.

1. We are given som A, c_1, c_2 such that $A \approx_{R'} (c_1 = c_2)$. So $A \approx_R (c_1 = c_2)$. Then by invariant 1 and monotonicity (lemma 56) we have $c_1 \approx_{E',R'} c_2$ as required.

2. We are given some $G\,\overline{c_i}$ and $G\,\overline{c_i}' \in E_0 \setminus E'$, and we need to show $G\,\overline{c_i} \approx_{E',R'} G\,\overline{c_i}'$.

   By assumption we have $\forall i. c_i \approx_{R'} c_i'$. So $\forall i. c_i \approx_R c_i'$.

If $G\,\overline{c_i} \not\equiv F\,\overline{a_i}$, then we must also have $G\,\overline{c_i} \in (E_0 \setminus E)$ (since only one equation was removed from E), and similarly for $G\,\overline{c_i}'$. So then by invariant 2 and monotonicity we have $G\,\overline{c_i} \approx_{E',R'} G\,\overline{c_i}'$ as required.

Otherwise, we are given $\forall i. a_i \approx_R c_i'$, and we need to prove $F\,\overline{a_i} \approx_{E',R'} F\,\overline{c_i}'$. From the premise to the rule we know $S(F, r(a_1), \ldots, r(a_n)) = (F\ b_1, \ldots, b_n = b)$, so by invariant 5 we know that there is some equation $F\,\overline{b_i} = b \in (E_0 \setminus E)$ where $b_k \approx_R a_k$. So by transitivity we have $b_k \approx_R a_k$. Then by invariant 2 we have $F\,\overline{b_i} \approx_{E,R} F\,\overline{c_i}'$, and by monotonicity (lemma 56) $F\,\overline{b_i} \approx_{E,R} F\,\overline{c_i}'$. By the definition of R we have $F\,\overline{a_i} = b \in R$. So by transitivity $F\,\overline{a_i} \approx b \approx F\,\overline{b_i} \approx F\,\overline{c_i}'$ as required.

3. We are given some injective G such that $G\,\overline{c_i} \approx_{R'} G\,\overline{c_i}'$ and we must show $c_k \approx_{E',R'} c_k'$. By invariant 3 we know $c_k \approx_{E,R} c_k'$. Then apply monotonicity (lemma 56).

4. Similar to the case for UPDATE1

5. For the first direction, suppose $G\,\overline{c_i} = c \in (E_0 \setminus E')$. The set $E_0 \setminus E$ contains all equations in $E_0 \setminus E'$ except for $F\,\overline{a_i} = a$. So there are two cases. If $(G\,\overline{c_i} = c) \not\equiv (F\,\overline{a_i} = a)$, then we also have $(G\,\overline{c_i} = c) \in (E_0 \setminus E)$. So we can use similar reasoning as in the corresponding case for the MERGE transition. Otherwise, if $(G\,\overline{c_i} = c) \equiv (F\,\overline{a_i} = a)$, then in the new state we have $S'(F, a_1, \ldots, a_n) = (F\,\overline{b_i} = b)$, and we need to prove $a_k \approx_{R'} b_k$ and $b \approx_{E',R'} a$. We get $a_k \approx_{R'} b_k$ from invariant 5 for the previous state, and we get $a \approx_{E',R'} b$ from the equation that this transition added.

   For the converse direction, suppose that $(G\,\overline{c_i} = c)$ is in the range of S'. Since $S = S'$ it is was already in the range of S, we reason similarly to the corresponding case for the MERGE transition.

6. We are given some $c \approx_{R'} (c_1 = c_2)$, and must show that some suitable $(c_1' = c_2') \in Q'(c)$.

   By lemma 53, the new equation from S' is either used or not. If not, we have $c \approx_R (c_1 = c_2)$, and get $(c_1 = c_2) \in Q(c)$ by invariant 6 for the previous state. Otherwise the equation was used, which can happen in two ways:

   - $c \approx_R F\,\overline{a_i}$ and $a = (c_1 = c_2)$. Then by invariant 6 for the previous state we have $(c_1 = c_2) \in Q(a)$, and hence in $Q'(c)$.
   - $c \approx_R a$ and $F\,\overline{a_i} \approx_R (c_1 = c_2)$. The only equations mentioning $F\,\overline{a_i}$ in R are those arising from $S(F, a_1, \ldots, a_n)$, so this can only happen in two ways. Either $(F\,\overline{a_i}) \equiv (c_1 = c_2)$, in which case the transition rule explicitly adds $(c_1 = c_2)$ to $Q'(a)$. Or else the transition was via b, i.e. we had $F\,\overline{a_i} \approx_R b \approx_R (c_1 = c_2)$. In this case we know $(c_1' = c_2') \in Q(b)$ from invariant 6 for the previous state, and hence it is also in $Q'(c)$.

7. Similar to invariant 6.

8. Similar to invariant 6.

9. Similar to the corresponding case for the UPDATE1 transition.

□

The invariants in lemma 57 shows that the equivalence relation $\approx_R$ constructed by the algorithm is "locally" complete: it satifies the congruence rule as long as the conclusion of the rule only contains subterms from the context E_0. In order to show that it is "globally" complete, we need to know that all provable equations are provable using only subterms of the problem. One way to do that is to use the notion of normal-form evidence terms which we introduced previously.

***Lemma* 58** (Completeness for normal-form evidence terms)**.** Suppose Γ is a context of the form described in lemma 51, and let $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$ be the initial state of the algorithm for Γ, and suppose $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0) \Longrightarrow^* (\cdot, r, Q, I, U, A, S)$. Then:

- If $\Gamma \vdash^{\mathsf{L}} pS : A = B$, then A and B are flat terms over Γ and $A \approx_R B$.
- If $\Gamma \vdash^{\mathsf{L}} pC : A = B$ and A and B are flat terms over Γ, then $A \approx_R B$.
- If $\Gamma \vdash^{\mathsf{L}} p^* : A = B$ and A and B are flat terms over Γ, then $A \approx_R B$.
- If $\Gamma \vdash^{\mathsf{L}} p^*_{\mathsf{R}} : A = B$ and A is a flat term over Γ, then B is a flat term over Γ and $A \approx_R B$.

provided that every use of assumptions h in the proofs pS,pC, p^* and p^*_{R} either refer to an assumption $h : A \in \Gamma$ where A is a flat term over Γ, or are of the form $h_{\triangleright\mathsf{refl}}$.

Proof. We proceed by induction on the structure of the given evidence term. The cases for pS are:

The evidence is $x_{\triangleright p^*_{\mathsf{R}}}$ From the premises to the rule we know we have $x : A \in \Gamma$ and $\Gamma \vdash^{\mathsf{L}} p^*_{\mathsf{R}} : A = (a = b)$. By the assumptions to there are two possibilities for A

Either A is a flat term over that context Then by the mutual IH for p^*_{R}, $a = b$ is flat as well (as required), and $A \approx_R (a = b)$. By invariant 1 we have $a \approx_R b$ as required.

Or else, $p^*_{\mathsf{R}} \equiv \mathsf{refl}$, so $A \equiv (a = b)$. By the definition of flat context (definition 47) A can one of three things: either a flat term (so this is a label application of the label "=", and a and b are atoms), or an equation between atoms (so a and b are atoms), or a application-atom equation (so a is a label application, and by virtue of this precise equation it is a flat term over Γ). In either of the three cases a and b are flat terms over Γ as required, and by invariant 1 we have $a \approx_R b$ as required.

The evidence is $(x_{\triangleright p^*_{\mathsf{R}}})^{-1}$ Similar to the above case we get $a \approx_R b$, and therefore $b \approx_R a$ by symmetry.

The evidence is $\mathsf{inj}\ i\ pS$ By the IH for pS, we know pS proves an equation between flat terms. Since the injectivity rule applies, they must be two label applications, $\Gamma \vdash^{\mathsf{L}} pS : F\,\overline{a_i} = F\,\overline{b_i}$. So the conclusion of the rule is an equation $a_k = b_k$ between two atoms, and atoms are flat over any context.

By the IH we also know $F\,\overline{a_i} \approx_R F\,\overline{b_i}$, so by invariant 3 we get $a_k \approx_R b_k$.

The evidence is a chain p^*_{LR} In other words, it is either a single term pS, which we dealt with in the previous cases, or it is is a chain starting and ending with a synthesizable term, that is p^*_{LR} is $pS; q^*; rS$. In the latter case we use the IHs for pS and rS to see that two two sides of the equation q^* are flat terms, appeal to the mutual IH for q^*, and use transitivity to chain together the three equations.

The only case for pC is when **the evidence term is a use of congruence,** $\mathsf{cong}\,F\,p_1\,..\,p_i$. The only rule that applies is CCPCONG, so the equation in the conclusion must be between two label applications, $F\,\overline{a_i} = F\,\overline{b_i}$. By assumption we know that they are flat terms over Γ, i.e. both label applications appear as left-hand sides of equations in Γ and all the a_i and b_i are atoms.

Since a_i and b_i are atoms they are per definition flat over Γ, so the IHs apply and give $a_i \approx_R b_i$.

The initial context E_0 contains all equations in Γ, in particular it contains the defining equations for $F\,\overline{a_i}$ and $F\,\overline{b_i}$. So by invariant 2 we get $F\,\overline{a_i} \approx_R F\,\overline{b_i}$ as required.

The cases for p^* are:

The empty chain (refl) We then have $a \approx_R a$ by reflexivity of $\approx$.

A chain consisting of a single term, p The evidence term p must be either a checkable are a synthesizable term, so we appeal to the corresponding mutual IH.

In the case when it is a pS,

A chain of length > 1 The definition of chains stipulates that there must never be two adjacent pCs, so we know that the either the first or the second evidence term in the chain is a pS. This is similar to the case for p^*_{LR} above.

The cases for p^*_{R} are similar to the case for p^*_{LR} above. □

***Lemma* 59** (Termination of the CC algorithm)**.** If $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$ is the initial state corresponding to some (flat) context Γ, there exists some final state with an empty list of pending equations such that $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0) \Longrightarrow^* (\cdot, r, Q, I, U, A, S)$.

Proof. Consider the (finite) set X of all flat terms occuring in Γ. The termination metric is the lexicographic order on (Number of equivalence classes on X induced by R)$\times$(Number of application-atom equations in E)$\times$(Number of atom-atom equations in E).

None of the rules can increase the number of equivalence classes. TRIVIAL leaves number of app-atom equations unchanged and decreases atom-atom equations. MERGE adds all kinds of equations, but reduces the number of equivalence classes. UPDATE1/2 adds atom-atom equations but decrease the number of app-atom equations. □

***Theorem* 60** (Correctness of the CC algorithm)**.** Suppose Γ is any context, Γ' is the flattened version of Γ, and $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0)$ is the initial state of the algorithm corresponding to Γ'. Then $(E_0, r_0, Q_0, I_0, U_0, A_0, S_0) \Longrightarrow^* (\cdot, r, Q, I, U, A, S)$, and for any atomic a and b, we have $\Gamma \vdash a = b$ iff $a \approx_R b$.

Proof. By lemma 59 we know the algorithm will terminate in a state with E empty. In that state, if a and b have the same r-representative then by lemma 52 invariants 2 and 6 we know $\Gamma \vdash a = b$.

Conversely, suppose that $\Gamma \vdash a = b$, so $\Gamma \vdash^{\mathsf{L}} p : a = b$ for some p. By lemma 51 we know that $\Gamma' \vdash^{\mathsf{L}} p' : a = b$ for some proof p' where every assumption is either a flat term or plain assumption $h_{\triangleright \mathsf{refl}}$. (We know that a and b are not changed by the flattening step since they were assumed to be atoms). By lemma 27 we have $\Gamma' \vdash^{\mathsf{L}} p^* : a = b$ for some p^*, and inspecting the proof of that lemma we see that p^* still obeys the restricton on assumptions. Then by lemma 58 we have $a \approx_R b$. □

Requiring a and b to be atoms is not a serious restriction: if we want to check some non-atomic terms a' and b' for equality we can pick fresh constants a and b, and add the equations $a = a'$ and $b = b'$ to the context. Also, checking whether $a \approx_R b$ is a cheap operation. Since they are both atoms, the wanted equation is true iff in the final state of the algorithm a and b are in the same union-find class (have the same r-representative).

$$\frac{\vdash \Gamma}{\vDash \emptyset : \Gamma = \Gamma}\text{EESAME} \qquad \frac{\begin{array}{l}\vDash \sigma : \Gamma = \Gamma' \\ \Gamma \vdash A : \mathsf{Type} \\ \Gamma' \vDash A' = \sigma\, A \\ \Gamma' \vdash v : A' = \sigma\, A\end{array}}{\vDash \sigma\,\{x_{\triangleright v}/x\} : \Gamma, x : A = \Gamma', x : A'}\text{EECONS}$$

Figure 18. Context equivalence

E. Proofs about the core language

E.1 Equivalent contexts

The next properties concern a new relation $\vDash \sigma : \Gamma = \Gamma'$, defined in Figure 18, which uses a substitution σ as the witness to the equivalence of two contexts.

***Lemma* 61** (Regularity for context equivalence)**.** If $\vDash \sigma : \Gamma = \Gamma'$, then $\vdash \Gamma$ and $\vdash \Gamma'$.

Proof. Induction on $\vDash \sigma : \Gamma = \Gamma'$. In the EESAME case we have this as a premise. In the EECONS case, we have $\Gamma \vdash A : \mathsf{Type}$ as a premise, and get $\Gamma' \vdash A' : \mathsf{Type}$ from regularity of the congruence closure relation (lemma 24). □

***Lemma* 62** (Variables in equivalent contexts)**.** If $y : C \in \Gamma$, and $\vDash \sigma : \Gamma = \Gamma'$, then there exists C' such that $y : C' \in \Gamma'$ and $\Gamma \vDash C' = \sigma\, C$.

Proof. Induction on $\vDash \sigma : \Gamma = \Gamma'$. The EESAME case is trivial.

In the EECONS case, the rule looks like

$$\frac{\begin{array}{l}\vDash \sigma : \Gamma = \Gamma' \\ \Gamma \vdash A : \mathsf{Type} \\ \Gamma' \vDash A' = \sigma\, A \\ \Gamma' \vdash v : A' = \sigma\, A\end{array}}{\vDash \sigma\,\{x_{\triangleright v}/x\} : \Gamma, x : A = \Gamma', x : A'}\text{EECONS}$$

There are two cases. If $x = y$, so $A = C$, then we can pick $C' := A'$, and we have $\Gamma' \vDash C' = \sigma\, C$ as a premise. By weakening (lemma 23) we have $\Gamma', x : A' \vDash C' = \sigma\, C$ as required.

If $x \neq y$, then $y : C \in \Gamma$, so by IH we have $y : C' \in \Gamma'$ with $\Gamma' \vDash C' = \sigma\, C$. Again, use weakening to get $\Gamma', x : A' \vDash C' = \sigma\, C$. □

***Lemma* 63** (Context conversion preserves erasure)**.** If $\vDash \sigma : \Gamma = \Gamma'$, then for any expression a we have $|\sigma\, a| = |a|$.

Proof. Examining the definition of $\vDash \sigma : \Gamma = \Gamma'$ we see that the substitution only adds type casts, which are erased. □

***Lemma* 64** (Context conversion for annotated language, var case)**.** If $x : A \in \Gamma$ and $\vDash \sigma : \Gamma = \Gamma'$, then $\Gamma' \vdash \sigma\, x : \sigma\, A$.

Proof. Induction on the length of Γ.

Γ **is empty** This contradicts the assumption that $x \in \Gamma$.

Γ **is** $\Gamma_0, y : B$ **for some** $y \neq x$ Then by considering the possible derivations of $\vDash \sigma : \Gamma = \Gamma'$ we know we have $\vDash \sigma_0 : \Gamma_0 = \Gamma'_0$ (and so on). By the IH we have $\Gamma'_0 \vdash \sigma_0\, x : \sigma_0\, A$. So by weakening (lemma 13) we have $\Gamma' \vdash \sigma_0\, x : \sigma_0\, A$. Since x is a bound variable we can pick it to not be in the domain of σ_0, and since $\Gamma_0 \vdash A : \mathsf{Type}$ we know $x \notin \mathsf{FV}\,(A)$. So the is equivalent to $\Gamma' \vdash \sigma\, x : \sigma\, A$.

Γ **is** $\Gamma_0, x : A$ By considering the possible derivations of $\vDash \sigma : \Gamma = \Gamma'$ we know that we must have

$$\frac{\begin{array}{l}\vDash \sigma : \Gamma = \Gamma' \\ \Gamma \vdash A : \mathsf{Type} \\ \Gamma' \vDash A' = \sigma\, A \\ \Gamma' \vdash v : A' = \sigma\, A\end{array}}{\vDash \sigma\,\{x_{\triangleright v}/x\} : \Gamma, x : A = \Gamma', x : A'}\text{EECONS}$$

So in particular we know $\sigma\, x$ is $x_{\triangleright v}$, which by TCAST has the type $\sigma\, A$.

□

***Lemma* 65** (Context conversion for annotated language)**.**
If $\Gamma \vdash a : A$ and $\vDash \sigma : \Gamma = \Gamma'$, then $\Gamma' \vdash \sigma\, a : \sigma\, A$.

Proof. Induction on $\Gamma \vdash a : A$.

TVAR By lemma 64.

TTYPE Trivial.

TPI The IH for A gives $\Gamma' \vdash \sigma\, A : \mathsf{Type}$.

By TCCREFL we have $\Gamma' \vDash \sigma\, A = \sigma\, A$, and it is easy to pick some identify proof v such that $\Gamma' \vdash v : \sigma\, A = \sigma\, A$. Then by EECONS, $\vDash \sigma\, \{x_{\triangleright v}/x\} : \Gamma, x : \sigma\, A = \Gamma', x : \sigma\, A$.

So by the IH, we get $\Gamma', x : \sigma\, A \vdash \sigma\, B : \mathsf{Type}$.

Now apply TPI to get $\Gamma' \vdash (x{:}\sigma\, A) \to\ \sigma\, B : \mathsf{Type}$ as required.

TIPI Similar to the previous case.

TREC By the IH we get $\Gamma' \vdash (x{:}\sigma\, A_1) \to\ \sigma\, A_2 : \mathsf{Type}$.

By reasoning similar to the TPI case we get

$$\vDash \sigma\, \{f_{\triangleright v_1}/f\}\, \{x_{\triangleright v_2}/x\} : \Gamma, f : (x{:}\sigma\, A_1) \to\ \sigma\, A_2, x : \sigma\, A_1 = \Gamma', f : (x{:}\sigma\, A_1) \to\ \sigma\, A_2, x : \sigma\, A_1$$

and hence by IH we get $\Gamma', f : (x{:}\sigma\, A_1) \to\ \sigma\, A_2, x : \sigma\, A_1 \vdash \sigma\, a : \sigma\, A_2$.

Now apply TREC to get $\Gamma' \vdash \mathsf{rec}\ f_{(x{:}\sigma\, A_1) \to\ \sigma\, A_2}\ x.\sigma\, a : (x{:}\sigma\, A_1) \to\ \sigma\, A_2$ as required.

TIREC Similar to the previous case.

TDAPP By the IHs for a and v we get $\Gamma' \vdash \sigma\, a : \sigma\, (x{:}A) \to\ B$ and $\Gamma' \vdash \sigma\, v : \sigma\, A$. Then apply TDAPP.

TAPP, TIDAPP, TEQ Similar to the previous case.

TJOINC By the IH we get $\Gamma' \vdash \sigma\, a = \sigma\, b : \mathsf{Type}$. Context equivalences preserve erasure (lemma 63), so $|\sigma\, a| = |a|$, and therefore we still have $|\sigma\, a| \rightsquigarrow^{i}_{\mathsf{cbv}} c$. Similarly, $|\sigma\, b| \rightsquigarrow^{i}_{\mathsf{cbv}} c$. Then apply TJOINC.

TJOINP Similar to the previous case.

TJINJDOM By the IH we have $\Gamma' \vdash \sigma\, v : ((x : \sigma\, A_1) \to\ \sigma\, B_1) = ((x : \sigma\, A_2) \to\ \sigma\, B_2)$. Then by TJINJDOM we do indeed have $\Gamma' \vdash \mathsf{join}_{\mathsf{injdom}\ \sigma\, v} : \sigma\, A_1 = \sigma\, A_2$

TJINJRNG, TJIINJDOM, TJIINJRNG. TINJEQ Similar to the previous case.

TJSUBST The IHs give $\forall k.\ \Gamma' \vdash \sigma\, v_k : \sigma\, a_k = \sigma\, b_k$ and $\Gamma' \vdash \sigma\, B : \mathsf{Type}$. Since context equivalence preserves erasure (lemma 63) the premise $|B| = |(\{a_1/x_1\} \ldots \{a_j/x_j\}\, c = \{b_1/x_1\} \ldots \{b_j/x_j\}\, c)|$ is unchanged. Then apply TJSUBST.

TCAST The IHs give $\Gamma' \vdash \sigma\, a : \sigma\, A$ and $\Gamma' \vdash \sigma\, v : \sigma\, A = \sigma\, B$ and $\Gamma' \vdash \sigma\, B : \mathsf{Type}$. Then by TCAST we have $\Gamma' \vdash (\sigma\, a)_{\triangleright \sigma\, v} : \sigma\, B$ as required.

□

***Lemma* 66** (Context conversion for congruence closure). If $\vDash \sigma : \Gamma = \Gamma'$, then $\Gamma \vDash a = b$ implies $\Gamma' \vDash \sigma\, a = \sigma\, b$.

Proof. By induction on $\Gamma \vDash a = b$. The cases are

TCCREFL By context conversion for the annotated language (lemma 65), we have $\Gamma' \vdash \sigma\, a : \sigma\, A$. Then apply TCCREFL again.

TCCERASURE By context conversion for the annotated language (lemma 65), $\sigma\, a$ and $\sigma\, b$ are well-typed in Γ'. And applying a context equivalence σ does not affect the erasure of a term (lemma 66). Then apply TCCERASURE again.

TCCSYM Direct by IH.

TCCTRANS Direct by IH.

TCCASSUMPTION The rule looks like

$$\frac{\Gamma \vDash C = (a = b) \qquad y : C \in \Gamma}{\Gamma \vDash a = b}$$

By the IH we know $\Gamma' \vDash \sigma\, C = \sigma\, (a = b)$.

By lemma 62 there exists $y : C' \in \Gamma'$ with $\Gamma' \vDash C' = \sigma\, C$. So by transitivity (TCCTRANS) we have $\Gamma' \vDash C = \sigma\, (a = b)$. Note that $\sigma\, (a = b) \equiv (\sigma\, a = \sigma\, b)$ Apply TCCASSUMPTION.

TCCCONGRUENCE The given rule looks like

$$\frac{\begin{array}{l}\Gamma \vdash A = B : \mathsf{Type} \quad \forall k.\ \Gamma \vDash a_k = b_k \\ |A = B| = |\{a_1/x_1\} \ldots \{a_j/x_j\}\, c = \{b_1/x_1\} \ldots \{b_j/x_j\}\, c|\end{array}}{\Gamma \vDash A = B}\ \text{TCCCONGRUENCE}$$

By IH we know $\forall k.\ \Gamma' \vDash \sigma\, a_k = \sigma\, b_k$.

By context conversion for the annotated language (lemma 65) we know $\Gamma' \vdash \sigma\, A = \sigma\, B : \mathsf{Type}$. And since context equivalences do not affect the erasure of terms (lemma 63) we still have

$$|\sigma\, A = \sigma\, B| = \{\sigma\, a_1/x_1\} \ldots \{\sigma\, a_j/x_j\}\, c = \{\sigma\, b_1/x_1\} \ldots \{\sigma\, b_j/x_j\}\, c.$$

Now apply TCCCONGRUENCE.

TCCINJDOM, TCCINJRNG, TCCIINJDOM, TCCIINJRNG,TCCINJEQ Direct by IH.

□

***Lemma* 67** (Symmetry of context equivalence)**.** If $\vDash \sigma : \Gamma = \Gamma'$, then there exists ρ such that $\vDash \rho : \Gamma' = \Gamma$.

Proof. By induction on the judgement $\vDash \sigma : \Gamma = \Gamma'$. The EESAME case is trivial.

In the EECONS case we are given

$$\frac{\begin{array}{l}\vDash \sigma : \Gamma = \Gamma' \\ \Gamma \vdash A : \mathsf{Type} \\ \Gamma' \vDash A' = \sigma\, A \\ \Gamma' \vdash v : A' = \sigma\, A\end{array}}{\vDash \sigma\, \{x_{\triangleright v}/x\} : \Gamma, x : A = \Gamma', x : A'}\,\text{EECONS}$$

By IH we have $\vDash \rho : \Gamma' = \Gamma$.

Using that ρ to apply context conversion (lemma 66) to the premise $\Gamma' \vDash A' = \sigma\, A$, we get $\Gamma \vDash \rho\, A' = \rho\,(\sigma\, A)$.

By regularity of the context equivalence relation (lemma 61) we know $\Gamma \vdash A : \mathsf{Type}$, and since context equivalence preserves erasure (lemma 63) we know $|\rho\,(\sigma\, A)| = |A|$. So by TCCERASURE, we have $\Gamma \vDash \rho\, A' = A$. By TCCSYM we get $\Gamma \vDash A = \rho\, A'$.

Furthermore, by lemma 38 this equation is witnessed by some value $\Gamma \vdash v : A = \rho\, A'$. Now pick $\rho\, \{x_{\triangleright v}/x\}$ as the witnessing substitution. □

***Lemma* 68** (Context equivalence symmetry is an inverse)**.** If $\vDash \sigma : \Gamma = \Gamma'$ and $\vDash \rho : \Gamma' = \Gamma$ and $\Gamma \vdash a : A$, then $\Gamma \vDash \rho\, \sigma\, a = a$.

Proof. Since the substitutions only change erased parts of the term (lemma 63), $|\rho\, \sigma\, a| = |a|$. And by applying context conversion (lemma 65) twice we have $\Gamma \vdash \rho\, \sigma\, a : \rho\, \sigma\, A$. So by TCCERASURE, $\Gamma \vDash \rho\, \sigma\, a = a$. □

***Lemma* 69** (Contexts are equivalent if they are equal up to erasure)**.** If $\vdash \Gamma$ and $\vdash \Gamma'$ and $|\Gamma| = |\Gamma'|$, then there exists σ such that $\vDash \sigma : \Gamma = \Gamma'$.

Proof. Induction of the length of the contexts. (We know that Γ and Γ' have the same length since they erase to the same thing).

- Two empty contexts are trivially equivalent.
- Suppose the contexts are $\Gamma, x : A$ and $\Gamma', x : A'$. By inversion on $\vdash \Gamma, x : A$ we get $\vdash \Gamma$ and $\Gamma \vdash A : \mathsf{Type}$, and similarly we get $\vdash \Gamma'$ and $\Gamma' \vdash A' : \mathsf{Type}$. And we know $|\Gamma| = |\Gamma'|$ and $|A| = |A'|$.

  By the IH we know that there exists some σ such that $\vDash \sigma : \Gamma = \Gamma'$. By context conversion (lemma 65) we have $\Gamma' \vdash \sigma\, A : \mathsf{Type}$. And since context equivalences do not affect erasure, the $|\sigma\, A| = |A|$. Thus, A' and $\sigma\, A$ are two well-typed terms which are equal up to erasure, so by TCCERASURE we have $\Gamma' \vDash A' = \sigma\, A$.

  Finally, picking the term $v = \mathsf{join}_{\leadsto_{\mathsf{cbv}} 0\, 0 : A' = \sigma\, A}$, we have $\Gamma' \vdash v : A' = \sigma\, A$.

  So applying EECONS we have

  $$\vDash \sigma\, \{x_{\triangleright v}/x\} : \Gamma, x : A = \Gamma', x : A'$$

  as we wanted to prove.

□

***Lemma* 70** (Context conversion for injrng)**.** If $\Gamma \vDash \mathsf{injrng}\, A\, \mathsf{for}\, v$ and $\vDash \sigma : \Gamma = \Gamma'$, then $\Gamma' \vDash \mathsf{injrng}\, \sigma\, A\, \mathsf{for}\, \sigma\, v$.

Proof. We only show the case when A is $(x : A_1) \to A_2$; the case when A is $\bullet(x : A_1) \to A_2$ is similar.

We are given that for all B_1, B_2, if $\Gamma \vDash ((x : A_1) \to A_2) = ((x : B_1) \to B_2)$ and $\Gamma \vdash v_0 : A_1 = B_1$ is the corresponding proof term, then $\Gamma \vDash \{v/x\}\, A_2 = \{v_{\triangleright v_0}/x\}\, B_2$. We must show that for all B_1', B_2', if $\Gamma' \vDash ((x : \sigma\, A_1) \to \sigma\, A_2) = ((x : B_1') \to B_2')$ and $\Gamma' \vdash v_0' : \sigma\, A_1 = B_1'$, then $\Gamma \vDash \{\sigma\, v/x\}\, \sigma\, A_2 = \{(\sigma\, v)_{\triangleright v_0'}/x\}\, B_2$.

So consider some B_1', B_2', v_0' satisfying the hypothesis.

Let ρ be such that $\vDash \rho : \Gamma' = \Gamma$ (using lemma 67).

Then by context conversion (lemma 66) we have $\Gamma \vDash ((x : \rho\, \sigma\, A_1) \to \rho\, \sigma\, A_2) = ((x : \rho\, B_1') \to \rho\, B_2')$. By lemma 68 and transitivity this equation is equivalent to $\Gamma \vDash ((x : A_1) \to A_2) = (x : \rho\, B_1') \to \rho\, B_2'$. Suppose the proof term for this equation is $\Gamma \vdash v_{00} : A_1 = \rho\, B_1'$. By assumption we have $\Gamma \vDash \{v/x\}\, A_2 = \{(\rho\, v)_{\triangleright v_{00}}/x\}\, \rho\, B_2'$. Now by context conversion again, $\Gamma' \vDash \sigma\, \{v/x\}\, A_2 = \sigma\, \{v_{\triangleright v_{00}}/x\}\, \rho\, B_2'$.

Since x was a bound variable, we can pick it so it is not in the domain of σ or ρ, so the above equation is equivalent to $\Gamma' \vDash \{\sigma\, v/x\}\, \sigma\, A_2 = \{\sigma\,((\rho\, v)_{\triangleright v_{00}})/x\}\, \sigma\, \rho\, B_2'$. Since σ and ρ cancel (lemma 68), this equation is equivalent to $\Gamma' \vDash \{\sigma\, v/x\}\, \sigma\, A_2 = \{(\sigma\, v)_{\triangleright \sigma\, v_{00}}/x\}\, B_2'$.

By inversion on $\Gamma \vDash ((x : A_1) \to\ A_2) = ((x : B_1') \to\ B_2')$ (lemmas 24 and 15) we know $\Gamma, x : B_1' \vdash B_2' : \mathsf{Type}$, so by substitution (lemma 16) we have $\Gamma \vdash \{(\sigma\, v)_{\triangleright v_0'}/x\}\, B_2 : \mathsf{Type}$. Then since $|\{(\sigma\, v)_{\triangleright \sigma\, v_{00}}/x\}\, B_2'| = |\{(\sigma\, v)_{\triangleright v_0'}/x\}\, B_2'|$, by TCCERASURE and TCCTRANS we have $\Gamma' \vDash \{\sigma\, v/x\}\, \sigma\, A_2 = \{(\sigma\, v)_{\triangleright v_0'}/x\}\, B_2'$ as required. □

F. Properties of injrng

***Lemma* 71** (injrng respects CC)**.** If $\Gamma \vDash$ injrng A for v and $\Gamma \vDash A = B$, then $\Gamma \vDash$ injrng B for v.

Proof. By transitivity, any type which is equal to B is also equal to A. □

***Lemma* 72** (injrng up to erasure of the value)**.** If $\Gamma \vDash$ injrng $(x : A) \to\ B$ for v and $\Gamma \vdash v' : A$ and $|v'| = |v|$, then $\Gamma \vDash$ injrng $(x : A) \to$ B for v'

Proof. Let $A1$, $B1$ such that $\Gamma \vDash (x : A) \to\ B = (x : A_1) \to\ B_1$ with the proof term $\Gamma \vdash v_0 : ((x : A) \to\ B) = ((x : A_1) \to\ B_1)$. We need to show $\Gamma \vDash \{v'/x\}\, B = \{v'_{\triangleright v_0}/x\}\, B_1$.

By the injrng assumption we have $\Gamma \vDash \{v/x\}\, B = \{v_{\triangleright v_0}/x\}\, B_1$. So by regularity (lemma 24) we have $\Gamma \vdash \{v/x\}\, B : \mathsf{Type}$ and $\Gamma \vdash \{v_{\triangleright v_0}/x\}\, B_1 : \mathsf{Type}$.

Also, by inversion on $\Gamma \vDash ((x : A_1) \to\ A_2) = ((x : B_1') \to\ B_2')$ (lemmas 24 and 15) we know $\Gamma, x : A_1 \vdash A_2 : \mathsf{Type}$ and $\Gamma, x : B_1' \vdash B_2' : \mathsf{Type}$, so by substitution (lemma 16) we have $\Gamma \vdash \{v'/x\}\, A_2 : \mathsf{Type}$ $\Gamma \vdash \{v'_{\triangleright v_0}/x\}\, B_2' : \mathsf{Type}$. So by TCCERASURE $\Gamma \vDash \{v/x\}\, A_2 = \{v'/x\}\, A_2$ and $\{v'_{\triangleright v_0}/x\}\, B_2' = \{v'/x\}\, B_2'$. Conclude by TCCTRANS. □

***Lemma* 73** (Instantiating injrng with a different value on the right)**.** If $\Gamma \vDash$ injrng $(x : A) \to\ B$ for v and $\Gamma \vDash (x : A) \to\ B = (x : A') \to\ B'$ and $\Gamma \vdash v' : A'$ and $|v'| = |v|$, then $\Gamma \vDash \{v/x\}\, B = \{v'/x\}\, B'$.

Proof. By the assumption $\Gamma \vDash$ injrng $(x : A) \to\ B$ for v we know that $\Gamma \vDash \{v/x\}\, B = \{v_{\triangleright v_0}/x\}\, B'$.

By inversion on $\Gamma \vDash ((x : A) \to\ B) = ((x : A') \to\ B')$ (lemmas 24 and 15) we know $\Gamma, x : A' \vdash B' : \mathsf{Type}$, so by substitution (lemma 16) we have $\Gamma \vdash \{v'/x\}\, B' : \mathsf{Type}$. We also know $|\{v_{\triangleright v_0}/x\}\, B'| = |\{v'/x\}\, B'|$, so by TCCERASURE we have $\Gamma \vDash \{v_{\triangleright v_0}/x\}\, B' = \{v'/x\}\, B'$.

Then by TCCTRANS, $\Gamma \vDash \{v/x\}\, B = \{v'/x\}\, B'$ as required. □

G. Proofs about elaboration

In general, in the following we will use primed metavariables for fully-elaborated core language environments and terms.

This lemma states that the elaboration algorithm produces output that type checks according to the core language and differs from the input only in the erasable parts of the term.

***Lemma* 74** (Soundness w.r.t. fully annotated typing)**.**

1. If $\vdash \Gamma'$ and $\Gamma' \mathrel{\vdash\!\!\!\!\cdot}\ a \Rightarrow a' : A'$, then $\Gamma' \vdash a' : A'$ and $|a| = |a'|$.
2. If $\vdash \Gamma'$ and $\Gamma' \vdash A' : \mathsf{Type}$ and $\Gamma' \mathrel{\vdash\!\!\!\!\cdot}\ a \Leftarrow A' \rightsquigarrow a'$, then $\Gamma' \vdash a' : A'$ and $|a| = |a'|$.
3. If $\mathrel{\vdash\!\!\!\!\cdot}\ \Gamma \rightsquigarrow \Gamma'$, then $\vdash \Gamma'$ and $|\Gamma| = |\Gamma'|$

Proof. Induction on the assumed typing derivations. The cases for $\Gamma \mathrel{\vdash\!\!\!\!\cdot}\ b \Rightarrow b' : B$ are:

EITYPE Trivial.

EIVAR Trivial.

EIPI By ih. $\Gamma' \vdash A' : \mathsf{Type}$ and $|A| = |A'|$. $\Gamma', x : A' \vdash B' : \mathsf{Type}$ and $|B| = |B'|$. Thus $\Gamma' \vdash (x : A') \to\ B' : \mathsf{Type}$ and $|(x : A) \to\ B| = |(x : A') \to\ B'|$.

EIIPI Similar to EIIPI.

EIDAPP

$$\frac{\begin{array}{c}\Gamma \mathrel{\vdash\!\!\!\!\cdot}\ a \Rightarrow a' : A_1 \quad \Gamma \mathrel{\vdash\!\!\!\!\cdot}\ A_1 =^? (x : A) \to B \rightsquigarrow v_1 \\ \Gamma \mathrel{\vdash\!\!\!\!\cdot}\ v \Leftarrow A \rightsquigarrow v' \quad \Gamma \vDash \mathsf{injrng}\,(x : A) \to\ B \text{ for } v'\end{array}}{\Gamma \mathrel{\vdash\!\!\!\!\cdot}\ a\ v \Rightarrow a'_{\triangleright v_1}\ v' : \{v'/x\}\, B}\ \text{EIDAPP}$$

By ih we have $\Gamma' \vdash a' : A_1$ where $|a| = |a'|$. By assumption 19 we have $\Gamma' \vdash v_1 : A_1 = ((x : A) \to B)$. By several inversions (lemma 15) of this judgement, we can conclude $\Gamma' \vdash (x : A) \to B : \mathsf{Type}$ and $\Gamma' \vdash A : \mathsf{Type}$. Therefore by casting, $\Gamma' \vdash a'_{\triangleright v_1} : (x : A) \to B$. Also by induction we have $\Gamma' \vdash v' : A$ and $|v| = |v'|$. Therefore $\Gamma' \vdash a'_{\triangleright v_1}\ v' : \{v'/x\}\, B$ and $|a'_{\triangleright v_1}\ v'| = |a\ v|$.

EIAPP and EIDIAPP Similar to EIDAPP.

EIEQ Directly by induction.

EIJJOINC By induction $|a = b| = |a' = b'|$ and $\Gamma' \vdash a' = b' : \mathsf{Type}$. Therefore, we know that the terms have the same erasure (i.e. $|a| = |a'|$ and $|b| = |b'|$) so the same premises can used in rule TJOINC.

EIJOINP Similar to EIJOINC.

EIANNOT By induction.

The cases for $\Gamma \Vdash a \Leftarrow A \rightsquigarrow a'$ are:

ECREC By assumption 19 we have $\Gamma' \vdash v_1 : A = ((x\!:\!A_1) \to A_2)$. By inversions of this judgement (lemma 15), $\Gamma' \vdash (x\!:\!A_1) \to A_2 : \mathsf{Type}$ and $\Gamma', x : A_1 \vdash A_2 : \mathsf{Type}$. By core language weakening $\Gamma', f : (x\!:\!A_1) \to A_2, x : A_1 \vdash A_2 : \mathsf{Type}$, so the induction hypothesis applies. Therefore $\Gamma', f : (x\!:\!A_1) \to A_2, x : A_1 \vdash a' : A_2$ and $|a| = |a'|$. By TREC, we have $\Gamma' \vdash \mathsf{rec}\ f_{(xA_1) \to A}\ x.a' : (x\!:\!A_1) \to A$, and by TCAST, we have $\Gamma' \vdash (\mathsf{rec}\ f_{(xA_1) \to A_2}\ x.a')_{\triangleright \mathsf{symm}\ v_1} : A$. Furthermore the erasures are equal.

ECIREC Similar to ECREC.

ECREFL By assumption (analogous to 19) we have $\Gamma' \vdash v_1 : A = (a = b)$. By inversion, $\Gamma' \vdash a = b : \mathsf{Type}$. By assumption 20, we also have $\Gamma' \vdash v : a = b$, and that $|v| = \mathsf{join}$. Therefore by TCAST we conclude that $\Gamma' \vdash v_{\triangleright \mathsf{symm}\ v_1} : A$ and that $|v_{\triangleright \mathsf{symm}\ v_1}| = |\mathsf{join}_\bullet|$.

ECINF We know that $\Gamma' \vdash B : \mathsf{Type}$. By induction we have that $\Gamma' \vdash a' : A$ where $|a| = |a'|$. That means $|a'_{\triangleright v_1}| = |a|$ By assumption 20, we have $\Gamma' \vdash v_1 : A = B$, therefore we can use TCAST to conclude $\Gamma' \vdash a'_{\triangleright v_1} : B$.

The cases for $\Vdash \Gamma \rightsquigarrow \Gamma'$ are:

EGNIL Trivial.

EGVAR By the IH we know $\vdash \Gamma'$. So by the mutual IH for $\Gamma' \Vdash A \Leftarrow \mathsf{Type} \rightsquigarrow A'$ we know $\Gamma' \vdash A' : \mathsf{Type}$, and therefore $\vdash \Gamma', x : A'$. Similarly, $|\Gamma, x : A| = |\Gamma', x : A'|$.

□

G.1 Checking is closed under CC

This next lemma says that the input type of the elaboration judgement can be replaced with an equivalent type (according to congruence closure) and elaboration will still succeed, producing a result that differs only in typing annotations.

***Lemma* 75** (Admissibility of CCAST in elaboration)**.**
If $\Gamma' \Vdash a \Leftarrow A' \rightsquigarrow a'$ and $\Gamma' \vDash A' = B'$, then $\Gamma' \Vdash a \Leftarrow B' \rightsquigarrow a''$ for some a'' such that $|a''| = |a'|$.

Proof. Case analysis on $\Gamma' \Vdash a \Leftarrow A' \rightsquigarrow a'$. Cases ECREC, ECIREC, ECREFL, ECSUBST, ECDCON, and ECCASE are all very similar, so we show just ECREC in detail.

Here, the assumed typing derivation looks like

$$\frac{\begin{array}{l}\Gamma \Vdash A =^? (x : A_1) \to A_2 \rightsquigarrow v_1 \\ \Gamma, f : (x\!:\!A_1) \to A_2, x : A_1 \Vdash a \Leftarrow A_2 \rightsquigarrow a' \\ \Gamma, f : (x\!:\!A_1) \to A_2, x : A_1 \vDash \mathsf{injrng}\,(x\!:\!A_1) \to A_2 \text{ for } x \\ \Gamma, f : (x\!:\!A_1) \to A_2 \Vdash (x\!:\!A_1) \to A_2 \Leftarrow \mathsf{Type} \rightsquigarrow A_0\end{array}}{\Gamma \Vdash \mathsf{rec}\ f\ x.a \Leftarrow A \rightsquigarrow (\mathsf{rec}\ f_{(xA_1) \to A_2}\ x.a')_{\triangleright \mathsf{symm}\ v_1}}\ \text{ECREC}$$

By assumption 18 we have $\Gamma \Vdash B =^? (x\!:\!A_1) \to A_2$. Then apply ECREC again. The elaborated term only differs in the proof used by the cast, $\mathsf{symm}\ v_1$, and this difference gets erased.

The rule ECINF instead relies on transitivity of $\vDash$. We have $\Gamma' \Vdash A \stackrel{?}{=} B \rightsquigarrow v_1$ as a premise of the rule and $\Gamma' \vDash A' = A$ as an assumption, so $\Gamma' \vDash A' = B$, and hence $\Gamma' \Vdash A' \stackrel{?}{=} B \rightsquigarrow v_2$ for some v_2. Then apply ECINF again; again the elaborated term only differs by the proof of the cast. □

G.2 Context conversion for elaboration

***Lemma* 76** (Context conversion for elaboration)**.** Suppose $\vDash \sigma : \Gamma = \Gamma'$. Then,

1. $\Gamma \vdash a \Rightarrow a' : A$ implies $\Gamma' \vdash a \Rightarrow a'' : A'$ for some A' such that $\Gamma' \vDash A' = \sigma\, A$ and some a'' such that $|a''| = |a'|$.
2. $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma \vdash a \Leftarrow A \rightsquigarrow a'$ implies $\Gamma' \vdash a \Leftarrow \sigma\, A \rightsquigarrow a''$ for some a'' such that $|a''| = |a'|$.

Proof. Induction on the assumed derivations. The cases for $\Gamma \vdash b \Rightarrow b' : B$ are:

EITYPE Pick $A' := \mathsf{Type}$.

EIVAR By the variable lookup lemma (lemma 62) we have $x : A' \in \Gamma$ with $\Gamma' \vDash A' = \sigma\, A$, as required. The elaborated term is still x, so it is equal up to erasure as required.

EIPI By the mutual IHs we have $\Gamma' \vdash A \Leftarrow \mathsf{Type} \rightsquigarrow A''$ and $\Gamma' \vdash B \Leftarrow \mathsf{Type} \rightsquigarrow B''$. Then re-apply EIPI. By IH the subterms of the elaborated term are equal up to erasure, so the entire elaborated term is also equal up to erasure.

EIIPI Similar to EIIPI.

EIAPP By the IH for the first premise we know $\Gamma' \vdash a \Rightarrow a'' : A_1'$ for some type A_1' such that $\Gamma' \vDash A_1' = \sigma\, A_1$.

From the premise $\Gamma \vdash A_1 =^? (x : A) \to B \rightsquigarrow v_1$ and context conversion (lemma 66) we get $\Gamma' \vDash \sigma\, A_1 = (x : \sigma\, A) \to \sigma\, B$. So by transitivity, $\Gamma' \vDash A_1' = (x : \sigma\, A) \to \sigma\, B$. So the search $\Gamma' \vdash A_1' =^? (x : A') \to B' \rightsquigarrow v_1'$ will succeed for some arrow type $(x : A') \to B'$ and proof v_1', since these exists at least one such arrow type.

Now note that by TCCTRANS and TCCINJDOM, we have $\Gamma' \vDash A' = \sigma\, A$. From the IH for b we know $\Gamma' \vdash b \Leftarrow \sigma\, A \rightsquigarrow b''$. So by casting the return type (lemma 75) we get $\Gamma' \vdash b \Leftarrow A' \rightsquigarrow b'''$.

Now apply EIAPP to get $\Gamma' \vdash a\ b \Rightarrow a''\ b''' : B'$. By TCCINJRNG we have $\Gamma' \vDash B' = \sigma\, B$ as required.

EIDAPP By the IH for the first premise, we know $\Gamma' \vdash a \Rightarrow a'' : A_1'$ for some type A_1' such that $\Gamma' \vDash A_1' = \sigma\, A_1$.

From the premise $\Gamma \vdash A_1 =^? (x : A) \to B \rightsquigarrow v_1$ and context conversion (lemma 66) we get $\Gamma' \vDash \sigma\, A_1 = (x : \sigma\, A) \to \sigma\, B$. So by transitivity, $\Gamma' \vDash A_1' = (x : \sigma\, A) \to \sigma\, B$. So the search $\Gamma' \vdash A_1' =^? (x : A') \to B' \rightsquigarrow v_1'$ will succeed for some arrow type $(x : A') \to B'$ and proof v_1', since these exists at least one such arrow type.

Now note that by TCCTRANS and TCCINJDOM, we have $\Gamma' \vDash A' = \sigma\, A$. From the IH for v we know $\Gamma' \vdash v \Leftarrow \sigma\, A \rightsquigarrow v''$. So by casting the return type (lemma 75) we get $\Gamma' \vdash v \Leftarrow A' \rightsquigarrow v'''$.

By context conversion for injrng (lemma 70) we get $\Gamma' \vDash \mathsf{injrng}\,(x : \sigma\, A) \to \sigma\, B$ for $\sigma\, v'$. Now by correctness of elaboration (lemma 74) we know $\Gamma' \vdash v''' : A'$ and also $|v| = |v'''|$. The latter also implies $|v'''| = |\sigma\, v'|$, so since injrng respects type equality and erasure (lemmas 71, 72) we then have $\Gamma' \vDash \mathsf{injrng}\,(x : A') \to B'$ for v'''

Then apply EIDAPP again, to get $\Gamma' \vdash a\ v \Rightarrow a'_{\triangleright v_1'}\ v''' : \{v'''/x\}\, B'$.

From $\Gamma' \vDash \mathsf{injrng}\,(x : A') \to B'$ for v''' we get $\Gamma' \vDash \{v'''/x\}\, B' = \{\sigma\, v'/x\}\, \sigma\, B$. Since we can pick the bound variable so that $x \notin \mathsf{FV}\,(B)$, that is the same as $\Gamma' \vDash \{v'''/x\}\, B' = \sigma\, \{v'/x\}\, B$, as required. Also as required, $|a'\ v'| = |a''\ v'''|$ since the subterms are equal up to erasure.

ottdrulenameEIdiapp Similar to EIDAPP.

EIEQ By the IHs we get $\Gamma' \vdash a \Rightarrow a'' : A_0$ and $\Gamma' \vdash b \Rightarrow b'' : B_0$, then apply EIEQ again.

EIJJOINC By the mutual IH we get $\Gamma' \vdash a = b \Leftarrow \mathsf{Type} \rightsquigarrow a'' = b''$. Since a' and a'' erase to the same thing we know $|a| \rightsquigarrow^i_{\mathsf{cbv}} c$ (and similarly for b''), so applying EIJOINC again we get $\Gamma' \vdash \mathsf{join}_{\rightsquigarrow_{\mathsf{cbv}}\, i\, j : a = b} \Rightarrow \mathsf{join}_{\rightsquigarrow_{\mathsf{cbv}}\, i\, j : a'' = b''} : a'' = b''$.

By soundness (lemma 74) and regularity (lemma 17) we know $a'' = b''$ is well-typed, so by TCCERASURE we have $\Gamma' \vDash (a' = b') = (a'' = b'')$ as required.

EIJOINP Similar to EIJOINC.

EIANNOT By the mutual IH we get $\Gamma' \vdash A \Leftarrow \mathsf{Type} \rightsquigarrow A''$ and $\Gamma' \vDash A'' = \sigma\, A'$. Again by mutual IH we have $\Gamma' \vdash a \Leftarrow \sigma\, A' \rightsquigarrow a''$. So by casting (lemma 75) we have $\Gamma \vdash a \Leftarrow A'' \rightsquigarrow a'''$.

Then apply EIANNOT again, to get $\Gamma \vdash a_A \Rightarrow a''' : A''$. We have $|a'''| = |a''| = |a'|$ as required.

The cases for $\Gamma \vdash a \Leftarrow A \rightsquigarrow a'$ are:

ECREC By context conversion for CC (lemma 66) we know $\Gamma' \vDash \sigma\, A = (x : \sigma\, A_1) \to \sigma\, A_2$. So the search $\Gamma' \vdash A =^? (x : A_1') \to A_2' \rightsquigarrow v_1'$ will succeed for some arrow type $(x : A_1') \to A_2'$ and proof v_1', since there exists at least one such arrow type.

By regularity of CC (lemma 24) and inversion for type well-formedness we know $\Gamma, x : A_1 \vdash A_2 : \mathsf{Type}$, and so by weakening (lemma 13) $\Gamma, f : (x : A_1) \to A_2, x : A_1 \vdash A_2 : \mathsf{Type}$. So the induction hypothesis for the a premise is available.

By TCCTRANS and TCCINJDOM, we have $\Gamma' \vDash (x : A'_1) \to A'_2 = (x : \sigma A_1) \to \sigma A_2$ and $\Gamma' \vDash A'_1 = \sigma A_1$. So $\vDash \sigma' : \Gamma, f : (x : A_1) \to A_2, x : A_1 = \Gamma', f : (x : A'_1) \to A'_2, x : A'_1$, where σ' is the substitution σ suitably extended. So by IH, $\Gamma', f : (x : A'_1) \to A_2, x : A'_1 \vdash a \Leftarrow \sigma' A_2 \leadsto a''$.

Because injrng respects context conversion (Lemma 70) we have $\Gamma', f : (x : \sigma A_1) \to \sigma A_2, x : \sigma A_1 \vDash \mathsf{injrng}\,(x : \sigma A_1) \to \sigma A_2\, \mathsf{for}\, \sigma\, x$. Since it respects CC (lemma 71) that implies $\Gamma', f : (x : \sigma A_1) \to \sigma A_2, x : \sigma A_1 \vDash \mathsf{injrng}\,(x : A'_1) \to A'_2\, \mathsf{for}\, \sigma\, x$. Also, using the CC judgements we proved above, we can construct a ρ such that $\vDash \rho : \Gamma', f : (x : \sigma A_1) \to \sigma A_2, x : \sigma A_1 = \Gamma', f : (x : A'_1) \to A'_2, x : A'_1$. So by lemma 70 again, we have $\Gamma', f : (x : A'_1) \to A'_2, x : A'_1 \vDash \mathsf{injrng}\,(x : \rho A'_1) \to \rho A'_2\, \mathsf{for}\, \rho\, \sigma\, x$. The variables f and x were bound, so we can pick them to not appear in the arrow type, so this is the same as $\Gamma', f : (x : A'_1) \to A'_2, x : A'_1 \vDash \mathsf{injrng}\,(x : A'_1) \to A'_2\, \mathsf{for}\, \rho\, \sigma\, x$. Finally, since injrng respects erasure (lemma 72) we can conclude that $\Gamma', f : (x : A'_1) \to A'_2, x : A'_1 \vDash \mathsf{injrng}\,(x : A'_1) \to A'_2\, \mathsf{for}\, x$.

By weakening of CC (lemma 23) we have $\Gamma', f : (x : A'_1) \to A'_2, x : A'_1 \vDash (x : A'_1) \to A'_2 = (x : \sigma A_1) \to \sigma A_2$. So by the injrng assumption we know that $\Gamma', f : (x : A'_1) \to A'_2, x : A'_1 \vDash A'_2 = \sigma A_2$.

So by casting (lemma 75) we have $\Gamma', f : (x : A'_1) \to A'_2, x : A'_1 \vdash a \Leftarrow A'_2 \leadsto a'''$.

Now apply ECREC to get $\Gamma' \vdash \mathsf{rec}\ f\ x.a \Leftarrow \sigma A \leadsto (\mathsf{rec}\ f\ x.a''')_{\triangleright \mathsf{symm}\ v'_1}$ as required.

ECIREC Similar to ECREC.

ECREFL By context conversion for CC (lemma 66) we know $\Gamma' \vDash \sigma A = \sigma (a = b)$. Therefore, $\Gamma' \vdash \sigma A =^? (a_1 = b_1) \leadsto v'_1$ will succeed for some $a_1 = b_1$ such that $\Gamma' \vDash \sigma (a = b) = (a_1 = b_1)$. By TCCINJEQ, that implies $\Gamma' \vDash \sigma\, a = a_1$ and $\Gamma' \vDash \sigma\, b = b_1$.

We know $\Gamma' \vDash (\sigma\, a) = (\sigma\, b)$ by context conversion for CC.

So by transitivity (TCCTRANS) we have $\Gamma' \vDash a_1 = b_1$.So $\Gamma' \vdash a_1 \stackrel{?}{=} b_1 \leadsto v'$ will also succeed.

Then apply ECREFL again. By assumption 18 we know $|v_{\triangleright \mathsf{symm}\ v_1}| = |v'_{\triangleright \mathsf{symm}\ v'_1}| = \mathsf{join}$, so the elaborated terms are equal up to erasure as required.

ECINF By the mutual IH we have $\Gamma' \vdash a \Rightarrow a'' : A'$ with $\Gamma' \vDash \sigma A = A'$. And by context conversion for CC (lemma 66) we have $\Gamma' \vDash \sigma A = \sigma B$. By transitivity, $\Gamma' \vDash A' = \sigma B$, so $\Gamma' \vdash A' \stackrel{?}{=} \sigma B \leadsto v'_1$ succeeds for some v'_1. Then apply ECINF again.

□

G.3 Completeness of elaboration

Note: in the following lemma statement and proof we use the convention that metavariables with primes ($A', B' \ldots$) are expressions in the fully annotated language, and metavariables without primes are in the surface language.

The first completeness lemma says that if the surface language CC judgement is derivable, then the target CC judgement is also derivable after elaborating the context and terms.

***Lemma* 77** (Completeness of CC)**.** If $\Gamma \vDash^{\exists} a = b$ and $\vdash \Gamma \leadsto \Gamma'$ and $\Gamma' \vdash a \Rightarrow a' : A'$ and $\Gamma' \vdash b \Rightarrow b' : B'$ then $\Gamma' \vDash a' = b'$

Proof. The proof follows from the fact that typing annotations don't matter to congruence closure (Lemma 44). By inversion of $\Gamma \vDash^{\exists} a = b$ we have some Γ'_1, a'_1 and b'_1 such that $\Gamma'_1 \vDash a' = b'$ and $|\Gamma'_1| = |\Gamma|$, $|a'_1| = |a|$, and $|b'_1| = |b|$. By translation soundness (Lemma 74), we also have $|\Gamma'| = |\Gamma|$, $|a'| = |a|$, and $|b'| = |b|$, with $\Gamma' \vdash a' : A'$ and $\Gamma' \vdash b' : B'$. This is all that we need to use the lemma. □

Likewise, we need to know that the surface language injrng judgement also describes when the corresponding fully annotated version is derivable.

***Lemma* 78** (Completeness of injrng)**.** If $\Gamma \vDash^{\exists} \mathsf{injrng}\,(x : A) \to B\, \mathsf{for}\, v$ and $\vdash \Gamma \leadsto \Gamma'$ and $\Gamma' \vdash (x : A) \to B \Leftarrow \mathsf{Type} \leadsto (x : A') \to B'$ and $\Gamma' \vdash v \Leftarrow A' \leadsto v'$ then $\Gamma' \vDash \mathsf{injrng}\,(x : A') \to B'\, \mathsf{for}\, v'$.

Proof. Consider A_1, B_1 such that $\Gamma' \vDash (x : A') \to B' = (x : A_1) \to B_2$ with the proof term $\Gamma' \vdash v_0 : ((x : A') \to B') = ((x : A_1) \to B_2)$. We must show $\Gamma' \vDash \{v'/x\}\, B' = \{v'_{\triangleright v_0}/x\}\, B_1$.

By inversion and substitution, we know that $\Gamma' \vdash \{v'/x\}\, B' : \mathsf{Type}$ and $\Gamma' \vdash \{v'_{\triangleright v_0}/x\}\, B_1 : \mathsf{Type}$.

Now instantiation the assumption $\Gamma \vDash^{\exists} \mathsf{injrng}\,(x : A) \to B\, \mathsf{for}\, v$ with A_1 and B_1. We have $\Gamma \vDash^{\exists} \{v_A/x\}\, B = \{v_{A_1}/x\}\, B_1$. That is, there are some Γ'', a'' and b'' such that $|\Gamma''| = |\Gamma|$ and $|a''| = |\{v_A/x\}\, B|$ and $|b''| = |\{v_{A_1}/x\}\, B_1|$ and $\Gamma'' \vDash a'' = b''$.

Since elaboration produced terms which are equal up to erasure, we also have $|\Gamma''| = |\Gamma'|$ and $|a''| = |\{v/x\}\, B'|$ and $|b''| = |\{v_{\triangleright v_0}/x\}\, B_1|$. So since CC doesn't care about annotations (lemma 44) we have $\Gamma' \vDash \{v'/x\}\, B' = \{v'_{\triangleright v_0}/x\}\, B_1$ as required. □

We next prove the completeness of the entire system using mutual induction on the three judgements of the surface language. For convenience, we use an alternative ("regularized") version of the typing rules, written $\Gamma \vdash_{\text{reg}} a \Rightarrow A$, that adds additional regularity assumptions to the typing judgement. For example, in the RIDAPP rule we add the premise $\Gamma \vdash (x : A) \to B \Leftarrow \mathsf{Type}$. The typing rules for that system are shown in Figure 19.

To justify the addition of these premises, we show the following regularity lemma about the inference judgement.

***Lemma* 79.** If $\Gamma \vdash a \Rightarrow A$ then $\Gamma \vdash A \Leftarrow \mathsf{Type}$.

Proof. Proof is by case analysis of $\Gamma \vdash a \Rightarrow A$.

ITYPE Holds by ITYPE and CINF.

IVAR Holds by premise of the rule.

IPI Holds by ITYPE and CINF.

IDAPP Holds by premise of the rule.

IIDAPP Holds by premise of the rule.

IAPP Holds by premise of the rule.

IEQ Holds by ITYPE and CINF.

IJOINC Holds by premise of the rule.

IJOINP Holds by premise of the rule.

IANNOT Holds by premise of the rule.

ICAST Holds by premise of the rule.

□

***Lemma* 80** (Completeness, with strengthened invariants)**.** 1. If $\vdash_{\text{reg}} \Gamma \Leftarrow$ then $\Vdash \Gamma \rightsquigarrow \Gamma'$.

2. If $\Gamma \vdash_{\text{reg}} a \Rightarrow A$ and $\vdash_{\text{reg}} \Gamma \Leftarrow$ and $\Vdash \Gamma \rightsquigarrow \Gamma'$ and $\Gamma' \Vdash A \Leftarrow \mathsf{Type} \rightsquigarrow A'$, then $\Gamma' \Vdash a \Rightarrow a' : A''$ and $\Gamma' \vDash A' = A''$

3. If $\Gamma \vdash_{\text{reg}} a \Leftarrow A$ and $\vdash_{\text{reg}} \Gamma \Leftarrow$ and $\Vdash \Gamma \rightsquigarrow \Gamma'$ and $\Gamma' \Vdash A \Leftarrow \mathsf{Type} \rightsquigarrow A'$, then $\Gamma' \Vdash a \Leftarrow A' \rightsquigarrow a'$.

Proof. Mutual induction on the derivations. The cases for $\Gamma \vdash_{\text{reg}} a \Rightarrow A$ are:

ITYPE Pick $A' := \mathsf{Type}$.

IVAR By soundness of elaboration (lemma 74) applied to the assumption $\Vdash \Gamma \rightsquigarrow \Gamma'$, there is some $x : A'' \in \Gamma'$ with $|A''| = |A|$ and $\Gamma' \vdash A'' : \mathsf{Type}$. By soundness of elaboration applied to the assumption $\Gamma' \Vdash A \Leftarrow \mathsf{Type} \rightsquigarrow A'$, we know $\Gamma' \vdash A' : \mathsf{Type}$.

Now by EIVAR we have $\Gamma' \Vdash x \Rightarrow x : A''$, and by TCCERASURE $\Gamma' \vDash A' = A''$ as required.

IPI We know $\Gamma' \Vdash \mathsf{Type} \Leftarrow \mathsf{Type} \rightsquigarrow \mathsf{Type}$. So by the mutual IH for the A premise, $\Gamma' \Vdash A \Leftarrow \mathsf{Type} \rightsquigarrow A'$.

Then by GVAR we have $\vdash \Gamma, x : A \Leftarrow$, and by GFVAR we have $\Vdash \Gamma, x : A \rightsquigarrow \Gamma', x : A'$. So by the mutual IH for the B premise, $\Gamma', x : A' \Vdash B \Leftarrow \mathsf{Type} \rightsquigarrow B'$.

Now apply EIPI to get $\Gamma' \Vdash (x : A) \to B \Rightarrow (x : A') \to B' : \mathsf{Type}$.

IIPI Similar to IPI.

IDAPP The given typing derivation looks like

$$\frac{\begin{array}{l}\Gamma \vdash_{\text{reg}} (x : A) \to B \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\text{reg}} a \Rightarrow (x : A) \to B \\ \Gamma \vdash_{\text{reg}} v \Leftarrow A \\ \Gamma \vDash^{\exists} \mathsf{injrng}\,(x : A) \to B \text{ for } v \\ \Gamma \vdash_{\text{reg}} \{v_A/x\}\, B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\text{reg}} a\ v \Rightarrow \{v_A/x\}\, B}\text{RIDAPP}$$

In the regularized type system, we have $\Gamma \vdash (x : A) \to B \Leftarrow \mathsf{Type}$ as a premise to the given rule. So by IH, $\Gamma' \Vdash (x : A) \to B \Leftarrow \mathsf{Type} \rightsquigarrow B_1'$ for some type B_1', where $\Gamma' \vDash (x : A) \to B = B_1'$. In fact there is only one rule for elaborating arrow types, so by inversion of that judgement, we get $\Gamma' \Vdash (x : A) \to B \Leftarrow \mathsf{Type} \rightsquigarrow (x : A') \to B'$, where B_1' is $(x : A') \to B'$ and $\Gamma' \Vdash A \Leftarrow \mathsf{Type} \rightsquigarrow A'$ and $\Gamma', x : A' \Vdash B \Leftarrow \mathsf{Type} \rightsquigarrow B'$. By soundness, this also means that $|(x : A) \to B| = |(x : A') \to B'|$.

From the IH for the a premise we know $\Gamma' \Vdash a \Rightarrow a' : A_0'$ with $\Gamma' \vDash A_0' = (x : A') \to B'$.

$$\boxed{\Gamma \vdash_{\mathrm{reg}} a \Rightarrow A}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} a \Rightarrow A \quad \Gamma \vDash^{\exists} A = B \\ \Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} a \Rightarrow B}\ \textsc{RICast}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} a \Leftarrow A\end{array}}{\Gamma \vdash_{\mathrm{reg}} a_A \Rightarrow A}\ \textsc{RIAnnot}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} a \Rightarrow A \\ \Gamma \vdash_{\mathrm{reg}} b \Rightarrow B\end{array}}{\Gamma \vdash_{\mathrm{reg}} a = b \Rightarrow \mathsf{Type}}\ \textsc{RIEq}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} a_1 = a_2 \Leftarrow \mathsf{Type} \\ |a_1| \rightsquigarrow_{\mathsf{p}}^{i} b \quad |a_2| \rightsquigarrow_{\mathsf{p}}^{j} b\end{array}}{\Gamma \vdash_{\mathrm{reg}} \mathsf{join}_{\rightsquigarrow_{\mathsf{p}} i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\ \textsc{RIJoinP}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} a_1 = a_2 \Leftarrow \mathsf{Type} \\ |a_1| \rightsquigarrow_{\mathsf{cbv}}^{i} b \quad |a_2| \rightsquigarrow_{\mathsf{cbv}}^{j} b\end{array}}{\Gamma \vdash_{\mathrm{reg}} \mathsf{join}_{\rightsquigarrow_{\mathsf{cbv}} i\, j : a_1 = a_2} \Rightarrow a_1 = a_2}\ \textsc{RIJoinC}$$

$$\frac{}{\Gamma \vdash_{\mathrm{reg}} \mathsf{Type} \Rightarrow \mathsf{Type}}\ \textsc{RIType}$$

$$\frac{\vdash_{\mathrm{reg}} \Gamma \Leftarrow \quad x : A \in \Gamma \quad \Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type}}{\Gamma \vdash_{\mathrm{reg}} x \Rightarrow A}\ \textsc{RIVar}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type} \\ \Gamma, x : A \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} (x : A) \to B \Rightarrow \mathsf{Type}}\ \textsc{RIPi}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} A \to B \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} a \Rightarrow A \to B \\ \Gamma \vdash_{\mathrm{reg}} b \Leftarrow A\end{array}}{\Gamma \vdash_{\mathrm{reg}} a\ b \Rightarrow B}\ \textsc{RIApp}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} (x : A) \to B \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} a \Rightarrow (x : A) \to B \\ \Gamma \vdash_{\mathrm{reg}} v \Leftarrow A \\ \Gamma \vDash^{\exists} \mathsf{injrng}\,(x : A) \to B \text{ for } v \\ \Gamma \vdash_{\mathrm{reg}} \{v_A / x\}\, B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} a\ v \Rightarrow \{v_A / x\}\, B}\ \textsc{RIDApp}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type} \\ \Gamma, x : A \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} \bullet(x : A) \to B \Rightarrow \mathsf{Type}}\ \textsc{RIIPi}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} \bullet(x : A) \to B \Leftarrow \mathsf{Type} \\ \Gamma \vdash_{\mathrm{reg}} a \Rightarrow \bullet(x : A) \to B \\ \Gamma \vdash_{\mathrm{reg}} v \Leftarrow A \\ \Gamma \vDash^{\exists} \mathsf{injrng}\ \bullet (x : A) \to B \text{ for } v \\ \Gamma \vdash_{\mathrm{reg}} \{v_A / x\}\, B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} a\ \bullet_v \Rightarrow \{v_A / x\}\, B}\ \textsc{RIIDApp}$$

$$\boxed{\Gamma \vdash_{\mathrm{reg}} a \Leftarrow A}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathrm{reg}} a \Leftarrow A \\ \Gamma \vDash A = B \\ \Gamma \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash_{\mathrm{reg}} a \Leftarrow B}\ \textsc{RCCast}$$

$$\frac{\Gamma \vdash_{\mathrm{reg}} a \Rightarrow A}{\Gamma \vdash_{\mathrm{reg}} a \Leftarrow A}\ \textsc{RCInf}$$

$$\frac{\Gamma \vDash^{\exists} a = b}{\Gamma \vdash_{\mathrm{reg}} \mathsf{join} \Leftarrow a = b}\ \textsc{RCRefl}$$

$$\frac{\begin{array}{l}\Gamma, f : (x : A_1) \to A_2, x : A_1 \vdash_{\mathrm{reg}} a \Leftarrow A_2 \\ \Gamma, f : (x : A_1) \to A_2 \vdash_{\mathrm{reg}} A_1 \Leftarrow \mathsf{Type} \\ \Gamma, f : (x : A_1) \to A_2, x : A_1 \vdash_{\mathrm{reg}} A_2 \Leftarrow \mathsf{Type} \\ \Gamma, f : (x : A_1) \to A_2, x : A_1 \vDash^{\exists} \mathsf{injrng}\,(x : A_1) \to A_2 \text{ for } x\end{array}}{\Gamma \vdash_{\mathrm{reg}} \mathsf{rec}\ f\ x.a \Leftarrow (x : A_1) \to A_2}\ \textsc{RCRec}$$

$$\frac{\forall A'\ B'.((\Gamma \vDash^{\exists} ((x : A) \to B) = ((x : A') \to B')) \text{ implies } \Gamma \vDash^{\exists} \{v/x\}\, B = \{v/x\}\, B')}{\Gamma \vDash^{\exists} \mathsf{injrng}\,(x : A) \to B \text{ for } v}\ \textsc{EIR}$$

$$\frac{\begin{array}{l}\Gamma, f : \bullet(x : A_1) \to A_2, x : A_1 \vdash a \Leftarrow A_2 \\ x \notin \mathsf{FV}\,(|a|) \\ \Gamma, f : \bullet(x : A_1) \to A_2, x : A_1 \vDash^{\exists} \mathsf{injrng}\ \bullet (x : A_1) \to A_2 \text{ for } x \\ \Gamma, f : \bullet(x : A_1) \to A_2 \vdash \bullet(x : A_1) \to A_2 \Leftarrow \mathsf{Type}\end{array}}{\Gamma \vdash \mathsf{rec}\ f\ \bullet.a \Leftarrow \bullet(x : A_1) \to A_2}\ \textsc{CIRec}$$

Figure 19. Bidirectional typing rules for surface language, with added extra regularity premises

So, by Assumption 21 the search $\Gamma' \mathrel{\scriptstyle\vdash} A'_0 =^{?} (x : A'') \rightarrow B'' \rightsquigarrow v_1$ through the equivalence class of A'_0 will terminate successfully with some arrow type $(x\!:\!A'') \rightarrow\ B''$ and proof v_1, since there exists at least one such arrow type, and by Assumption 19 we know that $\Gamma' \vdash v_1 : (A'_0 = ((x\!:\!A'') \rightarrow\ B''))$.

As a result, we have $\Gamma' \vDash (x\!:\!A') \rightarrow\ B' = (x\!:\!A'') \rightarrow\ B''$, By TCCINJDOM we know $\Gamma' \vDash A' = A''$.

Now by the IH for the v premise, we get $\Gamma' \mathrel{\scriptstyle\vdash} v \Leftarrow A' \rightsquigarrow v'$ and, by lemma 74, that $|v| = |v'|$. By casting (lemma 75) this implies $\Gamma' \mathrel{\scriptstyle\vdash} v \Leftarrow A'' \rightsquigarrow v''$. Again by soundness (lemma 74), we have $\Gamma' \vdash v'' : A''$ and $|v| = |v''|$.

The algorithmic injrng premise of EIDAPP, namely $\Gamma' \vDash \mathsf{injrng}\,(x\!:\!A'') \rightarrow\ B''\ \mathsf{for}\ v''$ is satisfied by Lemma 78.

Now apply EIDAPP, to get $\Gamma' \mathrel{\scriptstyle\vdash} a\ v \Rightarrow a'\ v'' : \{v''/x\}\, B''$.

We know by assumption that $\Gamma' \mathrel{\scriptstyle\vdash} \{v_A/x\}\, B \Leftarrow \mathsf{Type} \rightsquigarrow B_0$. The lemma also requires showing $\Gamma' \vDash B_0 = \{v''/x\}\, B''$. By instantiating the injrng premise at v' (lemma 73), it suffices to show that $\Gamma' \vDash B_0 = \{v'/x\}\, B'$. We derive this equality via TCCERASURE, as $\Gamma' \vdash B_0 : \mathsf{Type}$ (via soundness), $\Gamma' \vdash \{v'/x\}\, B' : \mathsf{Type}$ (via substation for annotated language), and $|B_0| = |\{v'/x\}\, B'|$. This last equality holds because, by $|B| = |B'|$ and $|v_A| = |v'|$ and the fact that substitution commutes with erasure we know that $|\{v_A/x\}\, B| = |\{v'/x\}\, B'|$. Furthermore by soundness, we have $|\{v_A/x\}\, B| = |B_0|$.

IIAPP, IAPP Similar to the previous case.

IEQ By the IHs for the (added) premises $\Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type}$ and $\Gamma \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}$, we know $\Gamma' \mathrel{\scriptstyle\vdash} A \Leftarrow \mathsf{Type} \rightsquigarrow A'$ and $\Gamma' \mathrel{\scriptstyle\vdash} B \Leftarrow \mathsf{Type} \rightsquigarrow B'$.

Then by the IHs for the premises for a and b we know $\Gamma' \mathrel{\scriptstyle\vdash} a \Rightarrow a' : A''$ and $\Gamma' \mathrel{\scriptstyle\vdash} b \Rightarrow b' : B''$. Now apply EIEQ to get $\Gamma' \mathrel{\scriptstyle\vdash} a = b \Rightarrow a' = b' : \mathsf{Type}$.

IJOINC,IJOINP By the IH for the premise $\Gamma \vdash a_1 = a_2 \Leftarrow \mathsf{Type}$ we know $\Gamma' \mathrel{\scriptstyle\vdash} a_1 = a_2 \Leftarrow \mathsf{Type} \rightsquigarrow A_0$. There is only one rule for elaborating equality types, so by inversion on that judgement we in fact have $\Gamma' \mathrel{\scriptstyle\vdash} a_1 = a_2 \Leftarrow \mathsf{Type} \rightsquigarrow a'_1 = a'_2$ and $\Gamma' \mathrel{\scriptstyle\vdash} a_1 \Rightarrow a'_1 : A'_1$ and $\Gamma' \mathrel{\scriptstyle\vdash} a_2 \Rightarrow a'_2 : A'_2$.

By soundness of elaboration 74 we know $|a_i| = |a'_i|$, so the the reduction behavior is the same. So apply EIJOINC to get $\Gamma' \mathrel{\scriptstyle\vdash} \mathsf{join}_{\rightsquigarrow_{\mathsf{cbv}} i\, j : a_1 = a_2} \Rightarrow \mathsf{join}_{\rightsquigarrow_{\mathsf{cbv}} i\, j : a'_1 = a'_2} : a'_1 = a'_2$. Also by soundness of elaboration we know $a'_1 = a'_2$ is well-typed, so by TCCREFL $\Gamma' \vDash (a'_1 = a'_2) = (a'_1 = a'_2)$ as required.

IANNOT By the IH for the premise $\Gamma \vdash A \Leftarrow \mathsf{Type}$ we get $\Gamma' \mathrel{\scriptstyle\vdash} A \Leftarrow \mathsf{Type} \rightsquigarrow A'$. Then by the IH for $\Gamma \vdash a \Leftarrow A$, we get $\Gamma' \mathrel{\scriptstyle\vdash} a \Leftarrow A' \rightsquigarrow a'$. Now by EIANNOT, $\Gamma' \mathrel{\scriptstyle\vdash} a_A \Rightarrow a' : A'$.

By soundness of elaboration 74 we know A' is well-typed, so $\Gamma' \vDash A' = A'$ as required.

ICAST By the IH for the (added) premise $\Gamma \vdash_{\mathrm{reg}} A \Leftarrow \mathsf{Type}$, we have $\Gamma' \mathrel{\scriptstyle\vdash} A \Leftarrow \mathsf{Type} \rightsquigarrow A'$ (and $|A| = |A'|$ by soundness). Then by the IH for $\Gamma \vdash a \Rightarrow A$, we know $\Gamma' \mathrel{\scriptstyle\vdash} a \Rightarrow a' : A''$ with $\Gamma' \vDash A' = A''$. Likewise, by the IH for the (added) premise $\Gamma \vdash_{\mathrm{reg}} B \Leftarrow \mathsf{Type}$, we have $\Gamma' \mathrel{\scriptstyle\vdash} B \Leftarrow \mathsf{Type} \rightsquigarrow B'$ (and $|B| = |B'|$ by soundness).

By the definition of $\vDash^{\exists}$ we know there are Γ_1, A_1, B_1 such that $\Gamma_1 \vDash A_1 = B_1$ where $|\Gamma_1| = |\Gamma|$,$|A_1| = |A|$ and $|B_1| = |B|$, so by lemma 44 we have $\Gamma' \vDash A' = B'$. So by TCCTRANS $\Gamma' \vDash A'' = B'$, as required.

The cases for $\Gamma \vdash a \Leftarrow A$ are:

CREC The rule is

$$\frac{\begin{array}{l}\Gamma, f : (x\!:\!A_1) \rightarrow\ A_2, x : A_1 \vdash_{\mathrm{reg}} a \Leftarrow A_2 \\ \Gamma, f : (x\!:\!A_1) \rightarrow\ A_2 \vdash_{\mathrm{reg}} A_1 \Leftarrow \mathsf{Type} \\ \Gamma, f : (x\!:\!A_1) \rightarrow\ A_2, x : A_1 \vdash_{\mathrm{reg}} A_2 \Leftarrow \mathsf{Type} \\ \Gamma, f : (x\!:\!A_1) \rightarrow\ A_2, x : A_1 \vDash^{\exists}\ \mathsf{injrng}\,(x\!:\!A_1) \rightarrow\ A_2\ \mathsf{for}\ x\end{array}}{\Gamma \vdash_{\mathrm{reg}} \mathsf{rec}\ f\ x.a \Leftarrow (x\!:\!A_1) \rightarrow\ A_2}\ \textsc{RCrec}$$

To apply the induction hypothesis to the first premise, we need to know how the type A_2 elaborates in both the context Γ' and in that context extended with f.

There is only one rule for elaborating arrow types. So by inversion on the hypothesis $\Gamma' \mathrel{\scriptstyle\vdash} (x\!:\!A_1) \rightarrow\ A_2 \Leftarrow \mathsf{Type} \rightsquigarrow A$, we in fact have $\Gamma' \mathrel{\scriptstyle\vdash} (x\!:\!A_1) \rightarrow\ A_2 \Leftarrow \mathsf{Type} \rightsquigarrow (x\!:\!A'_1) \rightarrow\ A'_2$ and $\Gamma' \mathrel{\scriptstyle\vdash} A_1 \Leftarrow \mathsf{Type} \rightsquigarrow A'_1$ and $\Gamma', x : A'_1 \mathrel{\scriptstyle\vdash} A_2 \Leftarrow \mathsf{Type} \rightsquigarrow A'_2$ for some A'_1 and A'_2 such that $\Gamma' \mathrel{\scriptstyle\vdash} A_1 \Leftarrow \mathsf{Type} \rightsquigarrow A'_1$ and $\Gamma', x : A'_1 \mathrel{\scriptstyle\vdash} A_2 \Leftarrow \mathsf{Type} \rightsquigarrow A'_2$.

Using the fact that $(x\!:\!A_1) \rightarrow\ A_2$ elaborates to $(x\!:\!A'_1) \rightarrow\ A'_2$, we know $\mathrel{\scriptstyle\vdash} \Gamma, f : (x\!:\!A_1) \rightarrow\ A_2 \rightsquigarrow \Gamma', f : (x\!:\!A'_1) \rightarrow\ A'_2$. So by the IH for the first regularity premise we have $\Gamma, f : (x\!:\!A'_1) \rightarrow\ A'_2 \mathrel{\scriptstyle\vdash} A_1 \Leftarrow \mathsf{Type} \rightsquigarrow A''_1$ for some A''_1.

Similarly, using that A_1 elaborates to A''_1 we know $\mathrel{\scriptstyle\vdash} \Gamma, f : (x\!:\!A_1) \rightarrow\ A_2, x : A_1 \rightsquigarrow \Gamma', f : (x\!:\!A'_1) \rightarrow\ A'_2, x : A''_1$. So by the IH for the second regularity premise we have $\Gamma, f : (x\!:\!A'_1) \rightarrow\ A'_2, x : A''_1 \mathrel{\scriptstyle\vdash} A_2 \Leftarrow \mathsf{Type} \rightsquigarrow A''_2$ for some A''_2.

Now by the IH for the first premise of the rule, we get $\Gamma, f : (x\!:\!A'_1) \rightarrow\ A'_2, x : A''_1 \mathrel{\scriptstyle\vdash} a \Leftarrow A''_2 \rightsquigarrow a'$ for some a'.

By soundness of elaboration we know $(x\!:\!A'_1) \rightarrow\ A'_2$ is a well-formed type, so $\Gamma' \vDash (x\!:\!A'_1) \rightarrow\ A'_2 = (x\!:\!A'_1) \rightarrow\ A'_2$. So the search $\Gamma' \mathrel{\scriptstyle\vdash} (x\!:\!A'_1) \rightarrow\ A'_2 =^{?} (x : A'''_1) \rightarrow A'''_2 \rightsquigarrow v_1$ will succeed for some arrow type $(x\!:\!A'''_1) \rightarrow\ A'''_2$, since there exists at least one.

By TCCINJDOM, we also know $\Gamma' \vDash A_1' = A_1'''$. Furthermore, by soundness of elaboration (lemma 74) we know that both A_1' and A_1'' are well-formed types in the context $\Gamma', f : (x : A_1') \to A_2'$, and that they erase to the same thing. So we have $\Gamma', f : (x : A_1') \to A_2' \vDash A_1' = A_1''$. By symmetry and transitivity, $\Gamma', f : (x : A_1') \to A_2' \vDash A_1''' = A_1''$. Let v_2 be a proof of that fact. Then using these two proofs, we can produce a proof of equivalence of the contexts.

$$\vDash \{f_{\triangleright v_1}/f\}\,\{x_{\triangleright v_2}/x\} : \Gamma', f : (x{:}A_1') \to A_2', x : A_1'' = \Gamma', f : (x{:}A_1''') \to A_2''', x : A_1'''.$$

So by context conversion (lemma 76) we know $\Gamma', f : (x{:}A_1''') \to A_2''', x : A_1''' \mapsto a \Leftarrow \{f_{\triangleright v_1}/f\}\,\{x_{\triangleright v_2}/x\}\,A_2' \rightsquigarrow a''$. By soundness of elaboration $\Gamma', x : A_1' \vdash A_2' : \mathsf{Type}$, so $f \notin \mathsf{FV}\,(A_2')$ and the above statement simplifies to $\Gamma', f : (x{:}A_1''') \to A_2''', x : A_1''' \mapsto a \Leftarrow \{x_{\triangleright v_2}/x\}\,A_2' \rightsquigarrow a''$.

By completeness of injrng (lemma 78), we know that $\Gamma', f : (x : A_1') \to A_2', x : A_1' \vDash \mathsf{injrng}\,(x : A_1') \to A_2'$ for x. By weakening the judgement $\Gamma' \vDash (x : A_1') \to A_2' = (x : A_1''') \to A_2'''$ (lemma 23), and because injrng respects CC (lemma 71), we get $\Gamma', f : (x : A_1') \to A_2', x : A_1' \vDash \mathsf{injrng}\,(x : A_1''') \to A_2'''$ for x. By the CC-equivalences proved above we can find a context equivalence $\vDash \rho : \Gamma', f : (x : A_1') \to A_2', x : A_1' = \Gamma', f : (x : A_1''') \to A_2''', x : A_1'''$. So since injrng respects context conversion (lemma 70) we have $\Gamma', f : (x{:}A_1''') \to A_2''', x : A_1''' \vDash \mathsf{injrng}\,\rho\,(x{:}A_1''') \to A_2'''$ for $\rho\,x$. We can pick the bound variables f and x to not be free in $(x{:}A_1') \to A_2'$, so this is the same as $\Gamma', f : (x{:}A_1''') \to A_2''', x : A_1''' \vDash \mathsf{injrng}\,(x{:}A_1''') \to A_2'''$ for $\rho\,x$. And because injrng respects erasure (lemma 72), we have $\Gamma', f : (x{:}A_1''') \to A_2''', x : A_1''' \vDash \mathsf{injrng}\,(x{:}A_1''') \to A_2'''$ for x, which is what we need as a premise to ECREC.

By TCCERASURE we know $\Gamma', x : A_1''' \vDash \{x_{\triangleright v_2}/x\}\,A_2' = A_2'''$. By weakening (lemma 23) thus $\Gamma', f : (x{:}A_1''') \to A_2''', x : A_1''' \vDash \{x_{\triangleright v_2}/x\}\,A_2' = A_2'''$. So by casting (lemma 75), we have $\Gamma', f : (x{:}A_1'') \to A_2'', x : A_1'' \mapsto a \Leftarrow A_2'' \rightsquigarrow a'''$.

Now apply ECREC to get $\Gamma \mapsto \mathsf{rec}\,f\ x.a \Leftarrow (x{:}A_1) \to A_2 \rightsquigarrow (\mathsf{rec}\,f_{(xA_1'')\to A_2''}\ x.a''')_{\triangleright \mathsf{symm}\ v_1}$.

CIREC Similar to CREC.

CREFL There is only one rule for elaborating equality types, so by inversion on the hypothesis $\Gamma' \mapsto a = b \Leftarrow \mathsf{Type} \rightsquigarrow A'$ we know that in fact $\Gamma' \mapsto a = b \Leftarrow \mathsf{Type} \rightsquigarrow a' = b'$ and $\Gamma' \mapsto a \Rightarrow a' : a_0'$ and $\Gamma' \mapsto b \Rightarrow b' : b_0'$. So by soundness of elaboration (lemma 74) we know a' and b' are well-typed terms, and therefore by lemma 44 and the premise $\Gamma \vDash^{\exists} a = b$, we have $\Gamma' \vDash a' = b'$.

So the search $\Gamma \mapsto (a' = b') =^{?} (a'' = b'') \rightsquigarrow v_1$ will terminate successfully with some equality type $a'' = b''$ such that $\Gamma' \mapsto a'' \stackrel{?}{=} b'' \rightsquigarrow v$, since there exists at least one such type.

Then apply ECREFL to get $\Gamma \mapsto \mathsf{join} \Leftarrow a' = b' \rightsquigarrow v_{\triangleright \mathsf{symm}\ v_1}$ as required.

CINF By the mutual IH we have $\Gamma \mapsto a \Rightarrow a' : A'$ for some A' such that $\Gamma \vDash A' = A$. By transitivity, $\Gamma \vDash A' = B$. Now apply ECINF.

□