Checking Equality-Saturation Merge and Extraction Certificates

Arthur Freitas Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

July 14, 2026

Abstract

Equality-saturation engines maintain equalities in an e-graph and later extract a representative from an e-class. This entry verifies an executable checker for the corresponding engine boundary. It directly parses the flat annotated S-expression format returned by egg 0.11.0's explanation API [9, 1]; flat explanations may apply numbered rewrite rules at positions or reuse equalities from earlier checked e-class merges. For extraction, a finite e-graph is represented as a topologically ordered e-class DAG. An executable bottom-up dynamic program selects a representative, and a formal proof establishes minimum additive cost over every term represented by the designated class. Class-aligned equality certificates additionally prove the selected term equivalent to the canonical class term.

The development reuses the term, substitution, position, context, and rewriting infrastructure of the AFP entries *First-Order Terms* and *First-Order Rewriting*. Its soundness statements target their existing conversion relation. It does not introduce another first-order term datatype, rewrite relation, generic equational proof calculus, or proof of Birkhoff's theorem. Those generic equational results and a checker for equality proofs already exist in IsaFoR/CeTA. The new material is the chronological e-graph merge discipline, positional reuse of recorded merges, certified e-class DAG semantics, and globally optimal dynamic-programming extraction.

Contents

1 Reused foundations

This entry is an extension of existing rewriting libraries, not an alternative foundation. It imports the AFP sessions `First_Order_Terms` and `First_Order_Rewriting` [4, 8]. In particular, it directly uses:

- the AFP datatype `term`, substitutions and substitution application;
- AFP positions, valid-position sets, subterms, contexts, `replace_at`, and executable position enumeration; and
- AFP rewrite rules, `rstep`, and the symmetric reflexive-transitive closure used for conversions.

IsaFoR's equational-reasoning theory already defines the `eq_proof` datatype with reflexivity, symmetry, transitivity, assumption, and congruence constructors; its executable `proves` function checks such proofs; and the same

theory relates provability, conversions, semantic entailment, and Birkhoff completeness [3]. CeTA also certifies equality proofs and completion certificates [6, 5, 7]. We therefore do not reproduce any of those definitions or results.

The AFP entry *Proof Terms for Term Rewriting* formalizes proof terms and their correspondence with multisteps [2]. Those proof terms represent reductions; the certificates here instead expose chronological e-class sharing and justify extraction over a finite represented DAG.

2 Difference from IsaFoR/CeTA

The generic equational content of a rule-only explanation is intentionally delegated to the AFP/IsaFoR foundation. The contribution here begins where an e-graph exposes data that a generic equational proof does not record:

1. The egg importer parses the public `get_flat_strings` format, locates its nested `Rewrite=>`/`Rewrite<=` annotation, resolves the named rule, and infers the substitution with the AFP matcher. The compiled step is then replayed independently.

2. A `Merge_At` step cites a concrete equality by its index in the e-graph's merge history and reuses it at an AFP position.

3. A merge log is checked chronologically. Entry i may cite only entries before i, which rules out circular e-class justifications.

4. A finite acyclic e-graph is represented by nonempty e-classes in topological order. Its semantics contains every e-node and every recursive combination of represented child terms.

5. An executable bottom-up dynamic program memoizes one optimum per class. The selected term is proved minimum-cost among all terms represented by the designated e-class.

6. Class-aligned flat certificates prove the canonical e-node equal to every alternative e-node. Congruence then proves every recursively represented term equal to the canonical class term.

7. A separate candidate-set checker supports interchange formats that enumerate terms explicitly.

8. A bounded producer accepts an arbitrary, untrusted step generator, but adds only steps accepted by the independent checker. Every emitted path is therefore a valid AFP conversion.

Thus this checker is not a replacement for CeTA's equational reasoning checker. It is an equality-saturation-specific front end whose accepted claims are reduced to the existing AFP conversion relation.

3 Fixture generated by egg

The end-to-end fixture uses egg 0.11.0 with its `deterministic` feature. The crates.io checksum is:

```
dd40cfd4196d7a8f882ace95d623d4d6734588e502b4e188675a7c2f55eb4fb4
```

The generator enabled explanations before adding the seed expression, disabled explanation-length optimization, ran the four named pattern rules shown in `Examples_Egg`, and executed:

```
# Cargo.toml
egg = { version = "=0.11.0", features = ["deterministic"] }

// Rust
let mut explanation =
    runner.explain_equivalence(&left, &right);
explanation.check_proof(&rules);
for step in explanation.get_flat_strings() {
    println!("{step}");
}
```

Repeated generator runs produced byte-identical output. The forward fixture consumed by Isabelle is:

```
(+ (shl x 1) (* y 2))
(+ (Rewrite=> shl-one (+ x x)) (* y 2))
(+ (+ x x) (Rewrite=> mul-two (+ y y)))
```

The backward fixture is:

```
z
(Rewrite<= mul-one (* z 1))
```

An additional regression fixture, `(f y (Rewrite=> add-zero z))`, checks that a rewrite in the second child is assigned AFP position [1] even when the first child is an atom. These strings are parsed during evaluation of the Isabelle checker; they are not manually replaced by preconstructed certificate values.

4 Verified interface

The main results are:

- `check_egg_explanation_sound`: an accepted textual flat explanation emitted by egg denotes an AFP conversion. The checked fixtures exercise both forward and backward annotations inside actual egg output.
- `check_explanation_sound`: an accepted flat explanation, relative to sound recorded merges, is an AFP conversion.
- `checked_merge_log_sound`: every equality in an accepted chronological merge log is an AFP conversion generated by the input rules.
- `egraph_explanation_sound`: a later explanation may safely reuse any entry of an accepted merge log.
- `represented_eclass_finite`: every class of a well-formed acyclic e-graph represents a finite set of terms.
- `extract_eclass_global_minimum`: the executable dynamic program returns a represented term no more expensive than any term represented by the designated class.
- `checked_egraph_eclass_sound`: every pair of terms represented by the same checked class is related by an AFP conversion.
- `certified_egraph_extraction_correct`: checked equality, representedness, and global minimum cost hold simultaneously for the extracted term.
- `run_bounded_search_accepted` and `find_explanation_sound`: every path returned by the bounded producer is accepted by the checker and denotes an AFP conversion, without assumptions on the proposal generator.

The acyclic-DAG example has a child class with three equivalent strength-reduction forms and a root e-node that references that class twice. The root therefore represents all nine child combinations. The checked dynamic program selects the globally cheapest one and proves it equal to the canonical root. The executable functions are exported to SML and Haskell. Another example embeds literal output generated by egg 0.11.0 with deterministic iteration and explanation-length optimization disabled. egg's own `check_proof` accepted the traces; the Isabelle checker then parses and independently replays them against AFP rewriting.

5 Scope

No union-find implementation, congruence-closure algorithm, rebuilding, e-matching, scheduling policy, or e-graph construction algorithm is verified. The global minimum ranges over all terms represented by the supplied finite acyclic DAG; it does not claim that an untrusted engine supplied every e-node it discovered. A separate candidate-set checker has only list-relative minimality. The DAG is ground, with source-language atoms represented as nullary symbols, as in egg's recursive expressions. The textual importer currently supports egg S-expressions with unquoted atoms and pattern-based named rules; custom appliers and quoted rule names require an adapter before checking.

6 Equality-saturation explanation certificates

theory *Equality-Saturation-Checker*
  imports
    First-Order-Rewriting.Trs
    First-Order-Terms.Term-Impl
begin

This development deliberately reuses the first-order terms, substitutions, positions, and contexts from the AFP session `First_Order_Terms`. It uses the rewrite relation from `First_Order_Rewriting`. In particular, the semantic target of the checker is the existing conversion relation $(rstep\ (set\ R))^{\leftrightarrow *}$; no separate term language or equational calculus is introduced here.

What is specific to equality saturation is the certificate boundary. A certificate is a flat path whose steps either apply a numbered input rule at an AFP position, or reuse a numbered equality recorded by an earlier checked e-class merge. Recorded merges are concrete term equalities, as in an e-graph, and therefore need no substitution when reused.

datatype *direction* = *Forward* | *Backward*

fun *orient-pair* :: *direction* ⇒ $'a \times 'a$ ⇒ $'a \times 'a$ **where**
  orient-pair Forward ab = *ab*
| *orient-pair Backward* (*a*, *b*) = (*b*, *a*)

datatype ($'f$, $'v$) *certificate-step* =
    Rule-At pos nat direction ($'f$, $'v$) *subst*
  | *Merge-At pos nat direction*

fun *apply-step* ::
  ($'f$, $'v$) *rule list* ⇒ ($'f$, $'v$) *rule list* ⇒
    ($'f$, $'v$) *certificate-step* ⇒ ($'f$, $'v$) *term* ⇒
    ($'f$, $'v$) *term option* **where**
  apply-step R Γ (*Rule-At p i d* σ) *t* =

```
    (if i < length R ∧ p ∈ poss t ∧
        t |- p = fst (orient-pair d (R ! i)) · σ
     then Some (replace-at t p (snd (orient-pair d (R ! i)) · σ))
     else None)
| apply-step R Γ (Merge-At p i d) t =
    (if i < length Γ ∧ p ∈ poss t ∧
        t |- p = fst (orient-pair d (Γ ! i))
     then Some (replace-at t p (snd (orient-pair d (Γ ! i))))
     else None)

fun apply-steps ::
  ('f, 'v) rule list ⇒ ('f, 'v) rule list ⇒
    ('f, 'v) certificate-step list ⇒ ('f, 'v) term ⇒
    ('f, 'v) term option where
  apply-steps R Γ [] t = Some t
| apply-steps R Γ (st # sts) t =
    (case apply-step R Γ st t of
       None ⇒ None
     | Some u ⇒ apply-steps R Γ sts u)

definition check-explanation ::
  ('f, 'v) rule list ⇒ ('f, 'v) rule list ⇒
    ('f, 'v) certificate-step list ⇒ ('f, 'v) term ⇒
    ('f, 'v) term ⇒ bool where
  check-explanation R Γ sts t u ⟷
    apply-steps R Γ sts t = Some u
```

6.1 Soundness in the AFP rewrite relation

```
lemma lift-conversion-at-position:
  assumes p: p ∈ poss t
    and sub: t |- p = s
    and conv: (s, u) ∈ (rstep R)^↔*
  shows (t, replace-at t p u) ∈ (rstep R)^↔*
proof −
  have ((ctxt-of-pos-term p t)⟨s⟩,
        (ctxt-of-pos-term p t)⟨u⟩) ∈ (rstep R)^↔*
    by (rule conversion-ctxt-closed[OF conv])
  moreover have (ctxt-of-pos-term p t)⟨s⟩ = t
    using ctxt-supt-id[OF p] sub by simp
  ultimately show ?thesis by simp
qed

lemma oriented-rule-conversion:
  assumes i: i < length R
  shows (fst (orient-pair d (R ! i)) · σ,
         snd (orient-pair d (R ! i)) · σ)
        ∈ (rstep (set R))^↔*
proof −
```

```
  obtain l r where lr: R ! i = (l, r) by (cases R ! i)
  have R ! i ∈ set R by (rule nth-mem[OF i])
  then have mem: (l, r) ∈ set R using lr by simp
  have step: (l · σ, r · σ) ∈ rstep (set R)
    by (rule rstep-subst[OF rstep-rule[OF mem]])
  show ?thesis
  proof (cases d)
    case Forward
    with lr step show ?thesis
      unfolding conversion-def by auto
  next
    case Backward
    with lr step show ?thesis
      unfolding conversion-def by auto
  qed
qed

lemma apply-step-sound:
  assumes merges:
    ∀ ab ∈ set Γ. (fst ab, snd ab) ∈ (rstep (set R))^↔*
    and step: apply-step R Γ st t = Some u
  shows (t, u) ∈ (rstep (set R))^↔*
proof (cases st)
  case (Rule-At p i d σ)
  from step Rule-At have i: i < length R and p: p ∈ poss t
    and sub: t |- p = fst (orient-pair d (R ! i)) · σ
    and u: u = replace-at t p (snd (orient-pair d (R ! i)) · σ)
    by (auto split: if-splits)
  have (fst (orient-pair d (R ! i)) · σ,
        snd (orient-pair d (R ! i)) · σ)
       ∈ (rstep (set R))^↔*
    by (rule oriented-rule-conversion[OF i])
  from lift-conversion-at-position[OF p sub this] u show ?thesis by simp
next
  case (Merge-At p i d)
  from step Merge-At have i: i < length Γ and p: p ∈ poss t
    and sub: t |- p = fst (orient-pair d (Γ ! i))
    and u: u = replace-at t p (snd (orient-pair d (Γ ! i)))
    by (auto split: if-splits)
  have mem: Γ ! i ∈ set Γ by (rule nth-mem[OF i])
  obtain a b where ab: Γ ! i = (a, b) by (cases Γ ! i)
  have base:
    (a, b) ∈ (rstep (set R))^↔*
    using merges mem ab by fastforce
  have oriented:
    (fst (orient-pair d (Γ ! i)), snd (orient-pair d (Γ ! i)))
      ∈ (rstep (set R))^↔*
  proof (cases d)
    case Forward
```

```
    with base ab show ?thesis by simp
  next
    case Backward
    have (b, a) ∈ (rstep (set R))^↔*
      using base
      by (rule conversion-sym[unfolded sym-def, rule-format])
    with Backward ab show ?thesis by simp
  qed
  from lift-conversion-at-position[OF p sub oriented] u show ?thesis by simp
qed

lemma apply-steps-sound:
  assumes merges:
    ∀ ab ∈ set Γ. (fst ab, snd ab) ∈ (rstep (set R))^↔*
    and run: apply-steps R Γ sts t = Some u
  shows (t, u) ∈ (rstep (set R))^↔*
  using run
proof (induction sts arbitrary: t)
  case Nil
  then show ?case by simp
next
  case (Cons st sts)
  from Cons.prems obtain v where
    first: apply-step R Γ st t = Some v and
    rest: apply-steps R Γ sts v = Some u
    by (auto split: option.splits)
  have tv: (t, v) ∈ (rstep (set R))^↔*
    by (rule apply-step-sound[OF merges first])
  have vu: (v, u) ∈ (rstep (set R))^↔*
    by (rule Cons.IH[OF rest])
  from tv vu show ?case
    unfolding conversion-def by (rule rtrancl-trans)
qed

theorem check-explanation-sound:
  assumes ∀ ab ∈ set Γ.
      (fst ab, snd ab) ∈ (rstep (set R))^↔*
    and check-explanation R Γ sts t u
  shows (t, u) ∈ (rstep (set R))^↔*
  using apply-steps-sound[OF assms(1) assms(2)[unfolded check-explanation-def]]
.

corollary check-rule-explanation-sound:
  assumes check-explanation R [] sts t u
  shows (t, u) ∈ (rstep (set R))^↔*
  by (rule check-explanation-sound[OF - assms]) simp

end
```

7 Importing egg flat explanations

theory *Egg-Flat-Explanation*
  imports
    Equality-Saturation-Checker
    First-Order-Terms.Matching
begin

The public explanation API of egg 0.11.0 emits a flat proof as one S-expression per term. The first term is unannotated. Every later term contains exactly one subexpression of the form (*Rewrite=> name term*) or (*Rewrite<= name term*). The former rewrites the previous term to the current term; the latter applies the named rule to the current term to recover the previous term.

This theory parses that actual textual format (for unquoted atoms), infers the substitution with the AFP matcher, compiles every annotation to *Rule-At*, and finally invokes the independently verified checker. Parsing and compilation are therefore outside the trusted claim: malformed or mistranslated input is accepted only if replay against the AFP rewrite rules succeeds.

datatype *egg-token* = *Egg-LParen* | *Egg-RParen* | *Egg-Atom string*

datatype *egg-sexp* = *Egg-SAtom string* | *Egg-SList egg-sexp list*

definition *egg-whitespace* :: *char set* **where**
  egg-whitespace =
    {*CHR '' ''*, *CHR ''⏎''*, *CHR 0x09*, *CHR 0x0D*}

definition *emit-egg-atom* ::
  string ⇒ *egg-token list* ⇒ *egg-token list* **where**
  emit-egg-atom atom toks =
    (*if atom* = [] *then toks else Egg-Atom* (*rev atom*) # *toks*)

fun *lex-egg* ::
  string ⇒ *string* ⇒ *egg-token list* ⇒ *egg-token list* **where**
  lex-egg [] *atom toks* = *rev* (*emit-egg-atom atom toks*)
| *lex-egg* (*c* # *cs*) *atom toks* =
    (*if c* ∈ *egg-whitespace then*
      lex-egg cs [] (*emit-egg-atom atom toks*)
     else if c = *CHR ''('' then*
      lex-egg cs [] (*Egg-LParen* # *emit-egg-atom atom toks*)
     else if c = *CHR '')'' then*
      lex-egg cs [] (*Egg-RParen* # *emit-egg-atom atom toks*)
     else
      lex-egg cs (*c* # *atom*) *toks*)

definition *tokenize-egg* :: *string* ⇒ *egg-token list* **where**
  tokenize-egg input = *lex-egg input* [] []

```
fun parse-egg-tokens ::
  egg-token list ⇒ egg-sexp list list ⇒ egg-sexp option where
  parse-egg-tokens [] stack =
    (case stack of
       [frame] ⇒
         (case rev frame of [x] ⇒ Some x | - ⇒ None)
     | - ⇒ None)
| parse-egg-tokens (tok # toks) stack =
    (case tok of
       Egg-Atom atom ⇒
         (case stack of
            [] ⇒ None
          | frame # frames ⇒
              parse-egg-tokens toks ((Egg-SAtom atom # frame) # frames))
     | Egg-LParen ⇒ parse-egg-tokens toks ([] # stack)
     | Egg-RParen ⇒
         (case stack of
            frame # parent # frames ⇒
              parse-egg-tokens toks
                ((Egg-SList (rev frame) # parent) # frames)
          | - ⇒ None))

definition parse-egg-sexp :: string ⇒ egg-sexp option where
  parse-egg-sexp input = parse-egg-tokens (tokenize-egg input) [[]]

fun plain-egg-term ::
  egg-sexp ⇒ (string, string) term option where
  plain-egg-term (Egg-SAtom atom) = Some (Fun atom [])
| plain-egg-term (Egg-SList xs) =
    (case xs of
       Egg-SAtom op-name # children ⇒
         (if op-name = ''Rewrite=>'' ∨ op-name = ''Rewrite<='' then
            (case children of
               [Egg-SAtom name, body] ⇒ plain-egg-term body
             | - ⇒ None)
          else map-option (Fun op-name)
            (Option-Monad.mapM plain-egg-term children))
     | - ⇒ None)

fun egg-annotations ::
  egg-sexp ⇒ (pos × direction × string) list
and egg-child-annotations ::
  nat ⇒ egg-sexp list ⇒
    (pos × direction × string) list where
  egg-annotations (Egg-SAtom atom) = []
| egg-annotations (Egg-SList xs) =
    (case xs of
       [Egg-SAtom op-name, Egg-SAtom name, body] ⇒
         (if op-name = ''Rewrite=>'' then
```

```
            ([], Forward, name) # egg-annotations body
          else if op-name = ''Rewrite<='' then
            ([], Backward, name) # egg-annotations body
          else egg-child-annotations 0 [Egg-SAtom name, body])
    | Egg-SAtom op-name # children ⇒
        egg-child-annotations 0 children
    | - ⇒ [])
| egg-child-annotations i [] = []
| egg-child-annotations i (x # xs) =
    map (λ(p, d, name). (i # p, d, name)) (egg-annotations x) @
    egg-child-annotations (Suc i) xs

record egg-mark =
  egg-pos :: pos
  egg-dir :: direction
  egg-rule-name :: string

record egg-decoded-line =
  egg-term :: (string, string) term
  egg-mark :: egg-mark option

definition decode-egg-line :: string ⇒ egg-decoded-line option where
  decode-egg-line input =
    (case parse-egg-sexp input of
       None ⇒ None
     | Some sexp ⇒
         (case plain-egg-term sexp of
            None ⇒ None
          | Some t ⇒
              (case egg-annotations sexp of
                 [] ⇒ Some (|egg-term = t, egg-mark = None|)
               | [(p, d, name)] ⇒
                   Some (|egg-term = t,
                     egg-mark = Some (|egg-pos = p, egg-dir = d,
                       egg-rule-name = name|)|)
               | - ⇒ None)))

type-synonym egg-named-rules =
  (string × (string, string) rule) list

fun named-rule-index ::
  string ⇒ egg-named-rules ⇒ nat option where
  named-rule-index name [] = None
| named-rule-index name ((other, r) # rules) =
    (if name = other then Some 0
     else map-option Suc (named-rule-index name rules))

definition compile-egg-step ::
  egg-named-rules ⇒ (string, string) term ⇒ string ⇒
```

```
    ((string, string) certificate-step × (string, string) term) option where
  compile-egg-step named previous input =
    (case decode-egg-line input of
       None ⇒ None
     | Some decoded ⇒
       (case egg-mark decoded of
          None ⇒ None
        | Some mark ⇒
          (case named-rule-index (egg-rule-name mark) named of
             None ⇒ None
           | Some i ⇒
             (if egg-pos mark ∈ poss previous then
                (case match (previous |- egg-pos mark)
                      (fst (orient-pair (egg-dir mark)
                        (map snd named ! i))) of
                   None ⇒ None
                 | Some σ ⇒
                   (let st = Rule-At (egg-pos mark) i
                       (egg-dir mark) σ
                    in if apply-step (map snd named) [] st previous =
                          Some (egg-term decoded)
                       then Some (st, egg-term decoded)
                       else None))
              else None))))

fun compile-egg-tail ::
  egg-named-rules ⇒ (string, string) term ⇒ string list ⇒
    ((string, string) certificate-step list ×
      (string, string) term) option where
  compile-egg-tail named current [] = Some ([], current)
| compile-egg-tail named current (line # lines) =
    (case compile-egg-step named current line of
       None ⇒ None
     | Some (st, next) ⇒
       (case compile-egg-tail named next lines of
          None ⇒ None
        | Some (sts, final) ⇒ Some (st # sts, final)))

fun compile-egg-explanation ::
  egg-named-rules ⇒ string list ⇒
    (((string, string) term × (string, string) term) ×
      (string, string) certificate-step list) option where
  compile-egg-explanation named [] = None
| compile-egg-explanation named (line # lines) =
    (case decode-egg-line line of
       None ⇒ None
     | Some decoded ⇒
       (case egg-mark decoded of
          Some mark ⇒ None
```

```
    | None ⇒
        (case compile-egg-tail named (egg-term decoded) lines of
          None ⇒ None
        | Some (sts, final) ⇒
            Some ((egg-term decoded, final), sts))))

definition check-egg-explanation ::
  egg-named-rules ⇒ string list ⇒
    (string, string) term ⇒ (string, string) term ⇒ bool where
  check-egg-explanation named lines expected-start expected-final =
    (case compile-egg-explanation named lines of
       None ⇒ False
     | Some ((start, final), sts) ⇒
         start = expected-start ∧ final = expected-final ∧
         check-explanation (map snd named) [] sts start final)

theorem check-egg-explanation-sound:
  assumes check-egg-explanation named lines start final
  shows (start, final)
    ∈ (rstep (set (map snd named)))^↔*
proof −
  from assms obtain sts where
    check-explanation (map snd named) [] sts start final
    unfolding check-egg-explanation-def
    by (auto split: option.splits prod.splits)
  then show ?thesis by (rule check-rule-explanation-sound)
qed

end
```

8 Explanations from checked e-class merges

```
theory EGraph-Explanations
  imports Equality-Saturation-Checker
begin
```

An equality-saturation engine accumulates equalities as it merges e-classes. Later explanations should be able to reuse those equalities without replaying the original rewrites. A merge log therefore stores each concrete equality together with a flat explanation that may cite only earlier entries. Checking the log from left to right prevents circular justification.

This is the main distinction from IsaFoR/CeTA's general equational proof checker. The certificate here represents an e-graph's chronological merge history and its reuse sites. All underlying rewriting remains the AFP relation *rstep*.

```
type-synonym ('f, 'v) merge-log =
  (('f, 'v) rule × ('f, 'v) certificate-step list) list
```

fun *check-merge-log-from* ::
  ('*f*, '*v*) *rule list* ⇒ ('*f*, '*v*) *rule list* ⇒
    ('*f*, '*v*) *merge-log* ⇒ *bool* **where**
  check-merge-log-from R Γ [] = *True*
| *check-merge-log-from R* Γ ((*ab*, *sts*) # *es*) =
    (*check-explanation R* Γ *sts* (*fst ab*) (*snd ab*) ∧
     check-merge-log-from R (Γ @ [*ab*]) *es*)

definition *check-merge-log* ::
  ('*f*, '*v*) *rule list* ⇒ ('*f*, '*v*) *merge-log* ⇒ *bool* **where**
  check-merge-log R log ⟷ *check-merge-log-from R* [] *log*

definition *recorded-merges* ::
  ('*f*, '*v*) *merge-log* ⇒ ('*f*, '*v*) *rule list* **where**
  recorded-merges log = *map fst log*

lemma *check-merge-log-from-sound*:
  assumes *check*: *check-merge-log-from R* Γ *es*
    and *base*: ∀ *ab* ∈ *set* Γ.
      (*fst ab*, *snd ab*) ∈ (*rstep* (*set R*))^↔*
  shows ∀ *ab* ∈ *set* (Γ @ *map fst es*).
      (*fst ab*, *snd ab*) ∈ (*rstep* (*set R*))^↔*
  using *check base*
proof (*induction es arbitrary*: Γ)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons e es*)
  obtain *ab sts* **where** *e*: *e* = (*ab*, *sts*) **by** (*cases e*)
  from *Cons.prems e* **have**
    cert: *check-explanation R* Γ *sts* (*fst ab*) (*snd ab*) **and**
    rest: *check-merge-log-from R* (Γ @ [*ab*]) *es*
    by *auto*
  have *ab-sound*:
    (*fst ab*, *snd ab*) ∈ (*rstep* (*set R*))^↔*
    by (*rule check-explanation-sound*[*OF Cons.prems*(*2*) *cert*])
  have *extended*: ∀ *ab'* ∈ *set* (Γ @ [*ab*]).
      (*fst ab'*, *snd ab'*) ∈ (*rstep* (*set R*))^↔*
    using *Cons.prems*(*2*) *ab-sound* **by** *auto*
  have ∀ *ab'* ∈ *set* ((Γ @ [*ab*]) @ *map fst es*).
      (*fst ab'*, *snd ab'*) ∈ (*rstep* (*set R*))^↔*
    by (*rule Cons.IH*[*OF rest extended*])
  moreover have (Γ @ [*ab*]) @ *map fst es* = Γ @ *map fst* (*e* # *es*)
    using *e* **by** *simp*
  ultimately show *?case* **by** *simp*
qed

theorem *checked-merge-log-sound*:

```
  assumes check-merge-log R log
    and ab ∈ set (recorded-merges log)
  shows (fst ab, snd ab) ∈ (rstep (set R))^↔*
proof −
  have all: ∀ ab ∈ set ([] @ map fst log).
      (fst ab, snd ab) ∈ (rstep (set R))^↔*
    by (rule check-merge-log-from-sound[
          OF assms(1)[unfolded check-merge-log-def]]) simp
  from assms(2) obtain entry where
    entry: entry ∈ set log and fst-entry: fst entry = ab
    unfolding recorded-merges-def by auto
  from all entry have
    (fst (fst entry), snd (fst entry))
      ∈ (rstep (set R))^↔*
    by simp
  with fst-entry show ?thesis by simp
qed

theorem egraph-explanation-sound:
  assumes log: check-merge-log R mlog
    and cert: check-explanation R (recorded-merges mlog) sts t u
  shows (t, u) ∈ (rstep (set R))^↔*
proof (rule check-explanation-sound[OF - cert])
  show ∀ ab ∈ set (recorded-merges mlog).
      (fst ab, snd ab) ∈ (rstep (set R))^↔*
  proof (intro ballI)
    fix ab
    assume ab ∈ set (recorded-merges mlog)
    then show (fst ab, snd ab) ∈ (rstep (set R))^↔*
      by (rule checked-merge-log-sound[OF log])
  qed
qed

end
```

9 Candidate-set extraction certificates

```
theory Extraction-Certificates
  imports EGraph-Explanations
begin
```

This lightweight interchange checker validates an explicitly enumerated candidate set: the chosen term is convertible to the source and has minimum cost among the supplied candidates. The stronger acyclic-e-graph development builds and verifies a dynamic-programming extractor over every term represented by a finite e-class DAG.

```
fun term-cost :: ('f ⇒ nat) ⇒ ('f, 'v) term ⇒ nat where
  term-cost w (Var x) = 1
```

```
| term-cost w (Fun f ts) = w f + sum-list (map (term-cost w) ts)

record ('f, 'v) extraction =
  ex-source :: ('f, 'v) term
  ex-chosen :: ('f, 'v) term
  ex-candidates ::
    (('f, 'v) term × ('f, 'v) certificate-step list) list

definition check-extraction ::
  ('f ⇒ nat) ⇒ ('f, 'v) rule list ⇒
    ('f, 'v) rule list ⇒ ('f, 'v) extraction ⇒ bool where
  check-extraction w R Γ E ⟷
     (∀ p ∈ set (ex-candidates E).
        check-explanation R Γ (snd p) (ex-source E) (fst p)) ∧
     ex-chosen E ∈ set (map fst (ex-candidates E)) ∧
     (∀ v ∈ set (map fst (ex-candidates E)).
        term-cost w (ex-chosen E) ≤ term-cost w v)

lemma check-extraction-candidate-sound:
  assumes merges: ∀ ab ∈ set Γ.
      (fst ab, snd ab) ∈ (rstep (set R))^↔*
    and check: check-extraction w R Γ E
    and candidate: v ∈ set (map fst (ex-candidates E))
  shows (ex-source E, v) ∈ (rstep (set R))^↔*
proof −
  from candidate obtain p where
    p: p ∈ set (ex-candidates E) and v: fst p = v
    by force
  from check p have
    check-explanation R Γ (snd p) (ex-source E) (fst p)
    unfolding check-extraction-def by blast
  from check-explanation-sound[OF merges this] v show ?thesis by simp
qed

lemma check-extraction-chosen-sound:
  assumes ∀ ab ∈ set Γ.
      (fst ab, snd ab) ∈ (rstep (set R))^↔*
    and check-extraction w R Γ E
  shows (ex-source E, ex-chosen E) ∈ (rstep (set R))^↔*
proof −
  from assms(2) have
    ex-chosen E ∈ set (map fst (ex-candidates E))
    unfolding check-extraction-def by simp
  from check-extraction-candidate-sound[OF assms this] show ?thesis .
qed

lemma check-extraction-minimal:
  assumes check-extraction w R Γ E
    and v ∈ set (map fst (ex-candidates E))
```

shows *term-cost w* (*ex-chosen E*) ≤ *term-cost w v*
using *assms* **unfolding** *check-extraction-def* **by** *blast*

theorem *extraction-over-checked-log-correct*:
assumes *log*: *check-merge-log R mlog*
and *extraction*:
check-extraction w R (*recorded-merges mlog*) *E*
shows (*ex-source E*, *ex-chosen E*)
∈ (*rstep* (*set R*))$^{\leftrightarrow *}$
and ∀ *v* ∈ *set* (*map fst* (*ex-candidates E*)).
term-cost w (*ex-chosen E*) ≤ *term-cost w v*
proof −
have *merges*: ∀ *ab* ∈ *set* (*recorded-merges mlog*).
(*fst ab*, *snd ab*) ∈ (*rstep* (*set R*))$^{\leftrightarrow *}$
using *checked-merge-log-sound*[*OF log*] **by** *blast*
show (*ex-source E*, *ex-chosen E*)
∈ (*rstep* (*set R*))$^{\leftrightarrow *}$
by (*rule check-extraction-chosen-sound*[*OF merges extraction*])
show ∀ *v* ∈ *set* (*map fst* (*ex-candidates E*)).
term-cost w (*ex-chosen E*) ≤ *term-cost w v*
using *extraction* **unfolding** *check-extraction-def* **by** *blast*
qed

end

10 Acyclic e-graph extraction

theory *Acyclic-EGraph-Extraction*
imports *Extraction-Certificates*
begin

A finite acyclic e-graph is represented by a list of e-classes in topological order. An e-node consists of a function symbol and indexes of child classes. Every child index must be smaller than the index of the containing class. Thus the list representation is both finite and a certificate of acyclicity. E-graph expressions are ground terms; atoms such as source-language variables are represented by nullary function symbols, as in egg's recursive-expression format.

The represented terms of a class include every e-node in that class and every combination of terms represented by its child classes. Extraction below is a bottom-up dynamic program: after finding an optimum for each earlier class, it instantiates every e-node with those optima and retains a minimum-cost result.

type-synonym *'f dag-enode* = *'f* × *nat list*
type-synonym *'f dag-eclass* = *'f dag-enode list*
type-synonym *'f acyclic-egraph* = *'f dag-eclass list*

```
definition wf-acyclic-egraph :: 'f acyclic-egraph ⇒ bool where
  wf-acyclic-egraph G ⟷
    (∀ i < length G.
      G ! i ≠ [] ∧
      (∀ node ∈ set (G ! i). ∀ j ∈ set (snd node). j < i))

inductive represents-eclass ::
  'f acyclic-egraph ⇒ nat ⇒ ('f, unit) term ⇒ bool
  for G where
  represents-node:
    i < length G ⟹
     (f, children) ∈ set (G ! i) ⟹
     list-all2 (represents-eclass G) children ts ⟹
     represents-eclass G i (Fun f ts)

definition instantiate-enode ::
  ('f, unit) term list ⇒ 'f dag-enode ⇒ ('f, unit) term
  where
  instantiate-enode memo node =
    Fun (fst node) (map ((!) memo) (snd node))

definition best-eclass-term ::
  ('f ⇒ nat) ⇒ ('f, unit) term list ⇒
    'f dag-eclass ⇒ ('f, unit) term
  where
  best-eclass-term w memo cls =
    instantiate-enode memo
      (arg-min-list (term-cost w ∘ instantiate-enode memo) cls)

fun extract-prefix ::
  ('f ⇒ nat) ⇒ 'f acyclic-egraph ⇒ nat ⇒
    ('f, unit) term list
  where
  extract-prefix w G 0 = []
| extract-prefix w G (Suc i) =
    (let memo = extract-prefix w G i
     in memo @ [best-eclass-term w memo (G ! i)])

definition extract-egraph ::
  ('f ⇒ nat) ⇒ 'f acyclic-egraph ⇒
    ('f, unit) term list option
  where
  extract-egraph w G =
    (if wf-acyclic-egraph G
     then Some (extract-prefix w G (length G))
     else None)

definition extract-eclass ::
  ('f ⇒ nat) ⇒ 'f acyclic-egraph ⇒ nat ⇒
```

```
    ('f, unit) term option
  where
  extract-eclass w G i =
    (if wf-acyclic-egraph G ∧ i < length G
     then Some (extract-prefix w G (length G) ! i)
     else None)

lemma extract-prefix-length [simp]:
  length (extract-prefix w G n) = n
  by (induction n) (simp-all add: Let-def)

lemma extract-prefix-nth-stable:
  assumes j < n
  shows extract-prefix w G n ! j = extract-prefix w G (Suc j) ! j
  using assms
proof (induction n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  show ?case
  proof (cases j = n)
    case True
    then show ?thesis by simp
  next
    case False
    with Suc.prems have j < n by simp
    then have j < length (extract-prefix w G n) by simp
    then have extract-prefix w G (Suc n) ! j = extract-prefix w G n ! j
      unfolding extract-prefix.simps Let-def
      by (rule nth-append-left)
    also have ... = extract-prefix w G (Suc j) ! j
      by (rule Suc.IH[OF ‹j < n›])
    finally show ?thesis .
  qed
qed

lemma arg-min-list-le:
  fixes score :: 'a ⇒ nat
  assumes xs ≠ [] and x ∈ set xs
  shows score (arg-min-list score xs) ≤ score x
proof −
  have score (arg-min-list score xs) = Min (score ‘ set xs)
    by (rule f-arg-min-list-f[OF assms(1)])
  also have ... ≤ score x
    by (rule Min-le) (use assms in auto)
  finally show ?thesis .
qed
```

```
lemma sum-list-mono-list-all2:
  fixes xs ys :: nat list
  assumes list-all2 (≤) xs ys
  shows sum-list xs ≤ sum-list ys
  using assms by (induction rule: list-all2-induct) simp-all

lemma finite-list-all2-fibres:
  assumes ∀ x ∈ set xs. finite {y. P x y}
  shows finite {ys. list-all2 P xs ys}
  using assms
proof (induction xs)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  have {ys. list-all2 P (x # xs) ys} =
      (λ(y, ys). y # ys) ` ({y. P x y} × {ys. list-all2 P xs ys})
    by (auto simp: list-all2-Cons1)
  moreover have finite ({y. P x y} × {ys. list-all2 P xs ys})
    using Cons by auto
  ultimately show ?case by simp
qed

theorem represented-eclass-finite:
  assumes wf: wf-acyclic-egraph G and i: i < length G
  shows finite {t. represents-eclass G i t}
proof −
  have finite-at:
    k < length G ⟹ finite {t. represents-eclass G k t} for k
  proof (induction k rule: less-induct)
    case (less k)
    have class-wf:
      G ! k ≠ [] ∧
        (∀ node ∈ set (G ! k). ∀ j ∈ set (snd node). j < k)
      using wf less.prems unfolding wf-acyclic-egraph-def by blast
    have fibres:
      finite {ts. list-all2 (represents-eclass G) children ts}
      if node: (f, children) ∈ set (G ! k) for f children
    proof (rule finite-list-all2-fibres)
      show ∀ j ∈ set children. finite {t. represents-eclass G j t}
      proof (intro ballI)
        fix j
        assume j ∈ set children
        with class-wf node have jk: j < k by auto
        with less.prems have j < length G by simp
        from less.IH[OF jk this] show finite {t. represents-eclass G j t} .
      qed
    qed
    have represented-union:
```

```
    {t. represents-eclass G k t} =
      (⋃(f, children) ∈ set (G ! k).
        Fun f ` {ts. list-all2 (represents-eclass G) children ts})
  proof (rule set-eqI, rule iffI)
    fix t
    assume t ∈ {t. represents-eclass G k t}
    then have rep: represents-eclass G k t by simp
    from rep obtain f children ts where
      t: t = Fun f ts and node: (f, children) ∈ set (G ! k) and
      related: list-all2 (represents-eclass G) children ts
      by (cases rule: represents-eclass.cases) auto
    from t node related show
      t ∈ (⋃(f, children) ∈ set (G ! k).
        Fun f ` {ts. list-all2 (represents-eclass G) children ts})
      by auto
  next
    fix t
    assume
      t ∈ (⋃(f, children) ∈ set (G ! k).
        Fun f ` {ts. list-all2 (represents-eclass G) children ts})
    then obtain f children ts where
      node: (f, children) ∈ set (G ! k) and
      related: list-all2 (represents-eclass G) children ts and
      t: t = Fun f ts
      by auto
    show t ∈ {t. represents-eclass G k t}
      using represents-eclass.represents-node[OF less.prems node related] t
      by simp
  qed
  show ?case unfolding represented-union using fibres by auto
 qed
 from finite-at[OF i] show ?thesis .
qed

lemma best-eclass-term-in:
 assumes cls ≠ []
 shows best-eclass-term w memo cls
   = instantiate-enode memo
       (arg-min-list (term-cost w ∘ instantiate-enode memo) cls)
   and arg-min-list (term-cost w ∘ instantiate-enode memo) cls
     ∈ set cls
 using assms unfolding best-eclass-term-def
 by (simp-all add: arg-min-list-in)

theorem extract-prefix-optimal:
 assumes wf: wf-acyclic-egraph G and i: i < length G
 shows
   represents-eclass G i (extract-prefix w G (Suc i) ! i)
   ⋀t. represents-eclass G i t ⟹
```

```
    term-cost w (extract-prefix w G (Suc i) ! i) ≤ term-cost w t
proof −
  have correct:
    k < length G ⟹
      represents-eclass G k (extract-prefix w G (Suc k) ! k) ∧
      (∀ t. represents-eclass G k t ⟶
        term-cost w (extract-prefix w G (Suc k) ! k) ≤ term-cost w t)
    for k
  proof (induction k rule: less-induct)
    case (less k)
    let ?memo = extract-prefix w G k
    let ?cls = G ! k
    let ?score = term-cost w ∘ instantiate-enode ?memo
    let ?best = arg-min-list ?score ?cls
    have class-wf:
      ?cls ≠ [] ∧
        (∀ node ∈ set ?cls. ∀ j ∈ set (snd node). j < k)
      using wf less.prems unfolding wf-acyclic-egraph-def by blast
    have best-mem: ?best ∈ set ?cls
      by (rule arg-min-list-in[OF class-wf[THEN conjunct1]])
    obtain f children where best: ?best = (f, children)
      by (cases ?best)
    have children-lt: ∀ j ∈ set children. j < k
      using class-wf best-mem best by auto
    have child-rep:
      list-all2 (represents-eclass G) children
        (map ((!) ?memo) children)
    proof (rule list-all2-all-nthI)
      show length children = length (map ((!) ?memo) children) by simp
      fix n
      assume n: n < length children
      let ?j = children ! n
      have j-mem: ?j ∈ set children using n by simp
      with children-lt have jk: ?j < k by blast
      with less.prems have jG: ?j < length G by simp
      have rep:
        represents-eclass G ?j (extract-prefix w G (Suc ?j) ! ?j)
        using less.IH[OF jk jG] by blast
      have ?memo ! ?j = extract-prefix w G (Suc ?j) ! ?j
        by (rule extract-prefix-nth-stable[OF jk])
      with rep show
        represents-eclass G (children ! n)
          (map ((!) ?memo) children ! n)
        using n by simp
    qed
    have chosen:
      extract-prefix w G (Suc k) ! k = instantiate-enode ?memo ?best
    proof −
      have len: k = length ?memo by simp
```

```
  have (?memo @ [instantiate-enode ?memo ?best]) ! k =
      instantiate-enode ?memo ?best
    using len by (metis nth-append-length)
  then show ?thesis by (simp add: Let-def best-eclass-term-def)
qed
have rep:
  represents-eclass G k (extract-prefix w G (Suc k) ! k)
proof −
  have node-best: (f, children) ∈ set ?cls
    using best-mem best by simp
  have represents-eclass G k
      (Fun f (map ((!) ?memo) children))
    by (rule represents-eclass.represents-node[
          OF less.prems node-best child-rep])
  then show ?thesis
    using chosen best unfolding instantiate-enode-def by simp
qed
have minimal:
  term-cost w (extract-prefix w G (Suc k) ! k) ≤ term-cost w t
  if t: represents-eclass G k t for t
proof −
  from t obtain g cs ts where
    t-eq: t = Fun g ts and node: (g, cs) ∈ set ?cls and
    related: list-all2 (represents-eclass G) cs ts
    using less.prems by (cases rule: represents-eclass.cases) auto
  have candidate-le:
    term-cost w (instantiate-enode ?memo ?best)
      ≤ term-cost w (instantiate-enode ?memo (g, cs))
    using arg-min-list-le[OF class-wf[THEN conjunct1] node, of ?score]
    by simp
  have child-costs:
    list-all2 (≤)
      (map (term-cost w) (map ((!) ?memo) cs))
      (map (term-cost w) ts)
  proof (rule list-all2-all-nthI)
    show length (map (term-cost w) (map ((!) ?memo) cs))
        = length (map (term-cost w) ts)
      using list-all2-lengthD[OF related] by simp
    fix n
    assume n: n < length (map (term-cost w) (map ((!) ?memo) cs))
    then have ncs: n < length cs by simp
    let ?j = cs ! n
    have j-mem: ?j ∈ set cs using ncs by simp
    from class-wf node j-mem have jk: ?j < k by auto
    with less.prems have jG: ?j < length G by simp
    have rep-child: represents-eclass G ?j (ts ! n)
      using list-all2-nthD[OF related ncs] .
    have le:
      term-cost w (extract-prefix w G (Suc ?j) ! ?j)
```

```
        ≤ term-cost w (ts ! n)
       using less.IH[OF jk jG] rep-child by blast
     have ?memo ! ?j = extract-prefix w G (Suc ?j) ! ?j
       by (rule extract-prefix-nth-stable[OF jk])
     with le show
       map (term-cost w) (map ((!) ?memo) cs) ! n
         ≤ map (term-cost w) ts ! n
       using ncs list-all2-lengthD[OF related] by simp
   qed
   have instantiated-le:
     term-cost w (instantiate-enode ?memo (g, cs)) ≤ term-cost w t
     using sum-list-mono-list-all2[OF child-costs]
     unfolding instantiate-enode-def t-eq by simp
   from candidate-le instantiated-le show ?thesis
     unfolding chosen by simp
 qed
 show ?case using rep minimal by blast
qed
from correct[OF i] show
 represents-eclass G i (extract-prefix w G (Suc i) ! i) by blast
from correct[OF i] show
 term-cost w (extract-prefix w G (Suc i) ! i) ≤ term-cost w t
 if represents-eclass G i t for t
 using that by blast
qed

theorem extract-eclass-global-minimum:
 assumes wf: wf-acyclic-egraph G and i: i < length G
 obtains t where
  extract-eclass w G i = Some t
  represents-eclass G i t
  ⋀u. represents-eclass G i u ⟹
    term-cost w t ≤ term-cost w u
proof −
 let ?t = extract-prefix w G (length G) ! i
 have stable:
  ?t = extract-prefix w G (Suc i) ! i
  by (rule extract-prefix-nth-stable[OF i])
 have extracted: extract-eclass w G i = Some ?t
  using wf i unfolding extract-eclass-def by simp
 have represented: represents-eclass G i ?t
  unfolding stable by (rule extract-prefix-optimal(1)[OF wf i])
 have minimal:
  term-cost w ?t ≤ term-cost w u
  if represents-eclass G i u for u
  unfolding stable by (rule extract-prefix-optimal(2)[OF wf i that])
 from that[OF extracted represented minimal] show thesis .
qed
```

end

11 Certified acyclic e-graphs

theory *Certified-Acyclic-EGraph*
  imports *Acyclic-EGraph-Extraction*
begin

The dynamic program establishes global optimality over the represented terms of an acyclic e-class DAG. This theory additionally checks that every e-node in a class is equal to a canonical e-node for that class.

Canonical terms are built bottom-up from the first e-node of every class. The certificate list for a class is aligned with its e-node list; each flat explanation must transform the canonical term into the corresponding e-node instantiated with canonical child terms. Congruence of AFP conversions then extends these finite certificates to every recursive combination represented by the DAG.

```
type-synonym 'f dag-class-certificates =
  ('f, unit) certificate-step list list

type-synonym 'f dag-certificates =
  'f dag-class-certificates list

fun canonical-prefix ::
  'f acyclic-egraph ⇒ nat ⇒ ('f, unit) term list
  where
  canonical-prefix G 0 = []
| canonical-prefix G (Suc i) =
    (let memo = canonical-prefix G i
     in memo @ [instantiate-enode memo (hd (G ! i))])

definition canonical-eclass ::
  'f acyclic-egraph ⇒ nat ⇒ ('f, unit) term option
  where
  canonical-eclass G i =
    (if wf-acyclic-egraph G ∧ i < length G
     then Some (canonical-prefix G (length G) ! i)
     else None)

definition check-dag-class ::
  ('f, unit) rule list ⇒ ('f, unit) term list ⇒
    'f dag-eclass ⇒ 'f dag-class-certificates ⇒ bool
  where
  check-dag-class R memo cls certs ⟷
    cls ≠ [] ∧
    list-all2
      (λnode sts.
```

```
      check-explanation R [] sts
        (instantiate-enode memo (hd cls))
        (instantiate-enode memo node))
    cls certs

definition check-certified-egraph ::
  ('f, unit) rule list ⇒ 'f acyclic-egraph ⇒
    'f dag-certificates ⇒ bool
  where
  check-certified-egraph R G certs ⟷
    wf-acyclic-egraph G ∧
    length certs = length G ∧
    list-all
      (λi. check-dag-class R (canonical-prefix G i)
        (G ! i) (certs ! i))
      [0..<length G]

lemma canonical-prefix-length [simp]:
  length (canonical-prefix G n) = n
  by (induction n) (simp-all add: Let-def)

lemma canonical-prefix-nth-stable:
  assumes j < n
  shows canonical-prefix G n ! j = canonical-prefix G (Suc j) ! j
  using assms
proof (induction n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  show ?case
  proof (cases j = n)
    case True
    then show ?thesis by simp
  next
    case False
    with Suc.prems have jn: j < n by simp
    then have j < length (canonical-prefix G n) by simp
    then have canonical-prefix G (Suc n) ! j = canonical-prefix G n ! j
      unfolding canonical-prefix.simps Let-def
      by (rule nth-append-left)
    also have ... = canonical-prefix G (Suc j) ! j
      by (rule Suc.IH[OF jn])
    finally show ?thesis .
  qed
qed

lemma canonical-prefix-step:
  canonical-prefix G (Suc i) ! i =
```

```
    instantiate-enode (canonical-prefix G i) (hd (G ! i))
proof −
  let ?memo = canonical-prefix G i
  have len: i = length ?memo by simp
  have (?memo @ [instantiate-enode ?memo (hd (G ! i))]) ! i =
      instantiate-enode ?memo (hd (G ! i))
    using len by (metis nth-append-length)
  then show ?thesis by (simp add: Let-def)
qed

lemma checked-egraph-wf:
  check-certified-egraph R G certs ⟹ wf-acyclic-egraph G
  unfolding check-certified-egraph-def by simp

lemma checked-dag-class-at:
  assumes checked: check-certified-egraph R G certs
    and i: i < length G
  shows check-dag-class R (canonical-prefix G i)
    (G ! i) (certs ! i)
  using checked i
  unfolding check-certified-egraph-def
  by (auto simp: list-all-iff)

lemma check-dag-class-certificate:
  assumes checked: check-dag-class R memo cls certs
    and node: node ∈ set cls
  obtains sts where
    check-explanation R [] sts
      (instantiate-enode memo (hd cls))
      (instantiate-enode memo node)
proof −
  from node obtain n where n: n < length cls and nth: cls ! n = node
    unfolding set-conv-nth by blast
  from checked have related:
    list-all2
      (λnode sts.
        check-explanation R [] sts
          (instantiate-enode memo (hd cls))
          (instantiate-enode memo node))
      cls certs
    unfolding check-dag-class-def by simp
  from list-all2-nthD[OF related n] nth have
    check-explanation R [] (certs ! n)
      (instantiate-enode memo (hd cls))
      (instantiate-enode memo node)
    by simp
  from that[OF this] show thesis .
qed
```

lemma *conversion-Fun-list-all2*:
assumes
list-all2
($\lambda s\ t.\ (s,\ t) \in (rstep\ R)^{\leftrightarrow *}$) *ss ts*
shows (*Fun f ss*, *Fun f ts*) $\in (rstep\ R)^{\leftrightarrow *}$
proof (*rule all-ctxt-closedD*[*OF all-ctxt-closed-rstep-conversion*])
show (*f*, *length ts*) $\in$ *UNIV* **by** *simp*
show *length ss* = *length ts*
using *list-all2-lengthD*[*OF assms*] **.**
fix *i*
assume $i < length\ ss$
from *list-all2-nthD*[*OF assms this*]
show (*ss* ! *i*, *ts* ! *i*) $\in (rstep\ R)^{\leftrightarrow *}$ **.**
qed *auto*

theorem *checked-egraph-representation-sound*:
assumes *checked*: *check-certified-egraph R G certs*
and *i*: $i < length\ G$
and *represented*: *represents-eclass G i t*
shows (*canonical-prefix G* (*Suc i*) ! *i*, *t*)
$\in (rstep\ (set\ R))^{\leftrightarrow *}$
using *i represented*
proof (*induction i arbitrary*: *t rule*: *less-induct*)
case (*less i*)
let *?memo* = *canonical-prefix G i*
let *?cls* = *G* ! *i*
from *less.prems*(*2*) **obtain** *f children ts* **where**
t: *t* = *Fun f ts* **and** *node*: (*f*, *children*) $\in$ *set ?cls* **and**
related: *list-all2* (*represents-eclass G*) *children ts*
using *less.prems*(*1*) **by** (*cases rule*: *represents-eclass.cases*) *auto*
have *wf*: *wf-acyclic-egraph G*
by (*rule checked-egraph-wf*[*OF checked*])
have *children-lt*: $\forall j \in set\ children.\ j < i$
using *wf less.prems*(*1*) *node* **unfolding** *wf-acyclic-egraph-def* **by** *auto*
have *canonical-children*:
list-all2
($\lambda s\ u.\ (s,\ u) \in (rstep\ (set\ R))^{\leftrightarrow *}$)
(*map* ((!) *?memo*) *children*) *ts*
proof (*rule list-all2-all-nthI*)
show *length* (*map* ((!) *?memo*) *children*) = *length ts*
using *list-all2-lengthD*[*OF related*] **by** *simp*
fix *n*
assume *n*: $n < length\ (map\ ((!)\ ?memo)\ children)$
then have *nc*: $n < length\ children$ **by** *simp*
let *?j* = *children* ! *n*
have *j-mem*: *?j* $\in$ *set children* **using** *nc* **by** *simp*
with *children-lt* **have** *ji*: $?j < i$ **by** *blast*
with *less.prems*(*1*) **have** *jG*: $?j < length\ G$ **by** *simp*
have *child-rep*: *represents-eclass G ?j* (*ts* ! *n*)

```
    using list-all2-nthD[OF related nc] .
  have conv:
    (canonical-prefix G (Suc ?j) ! ?j, ts ! n)
      ∈ (rstep (set R))^↔*
    by (rule less.IH[OF ji jG child-rep])
  have ?memo ! ?j = canonical-prefix G (Suc ?j) ! ?j
    by (rule canonical-prefix-nth-stable[OF ji])
  with conv show
    (map ((!) ?memo) children ! n, ts ! n)
      ∈ (rstep (set R))^↔*
    using nc by simp
qed
have node-to-term:
  (instantiate-enode ?memo (f, children), t)
    ∈ (rstep (set R))^↔*
proof −
  have (Fun f (map ((!) ?memo) children), Fun f ts)
      ∈ (rstep (set R))^↔*
    by (rule conversion-Fun-list-all2[OF canonical-children])
  then show ?thesis
    using t unfolding instantiate-enode-def by simp
qed
have class-checked:
  check-dag-class R ?memo ?cls (certs ! i)
  by (rule checked-dag-class-at[OF checked less.prems(1)])
from check-dag-class-certificate[OF class-checked node]
obtain sts where cert:
  check-explanation R [] sts
    (instantiate-enode ?memo (hd ?cls))
    (instantiate-enode ?memo (f, children)) .
have class-to-node:
  (instantiate-enode ?memo (hd ?cls),
    instantiate-enode ?memo (f, children))
    ∈ (rstep (set R))^↔*
  by (rule check-rule-explanation-sound[OF cert])
from class-to-node node-to-term have
  (instantiate-enode ?memo (hd ?cls), t)
    ∈ (rstep (set R))^↔*
  unfolding conversion-def by (rule rtrancl-trans)
then show ?case unfolding canonical-prefix-step .
qed

theorem checked-egraph-eclass-sound:
  assumes checked: check-certified-egraph R G certs
    and i: i < length G
    and s: represents-eclass G i s
    and t: represents-eclass G i t
  shows (s, t) ∈ (rstep (set R))^↔*
proof −
```

```
  have source-s:
    (canonical-prefix G (Suc i) ! i, s)
      ∈ (rstep (set R))^↔*
    by (rule checked-egraph-representation-sound[OF checked i s])
  have source-t:
    (canonical-prefix G (Suc i) ! i, t)
      ∈ (rstep (set R))^↔*
    by (rule checked-egraph-representation-sound[OF checked i t])
  have s-source:
    (s, canonical-prefix G (Suc i) ! i)
      ∈ (rstep (set R))^↔*
    using source-s
    by (rule conversion-sym[unfolded sym-def, rule-format])
  from s-source source-t show ?thesis
    unfolding conversion-def by (rule rtrancl-trans)
qed

theorem certified-egraph-extraction-correct:
  assumes checked: check-certified-egraph R G certs
    and i: i < length G
  obtains source chosen where
    canonical-eclass G i = Some source
    extract-eclass w G i = Some chosen
    (source, chosen) ∈ (rstep (set R))^↔*
    represents-eclass G i chosen
    ⋀u. represents-eclass G i u ⟹
      term-cost w chosen ≤ term-cost w u
proof −
  have wf: wf-acyclic-egraph G
    by (rule checked-egraph-wf[OF checked])
  let ?source = canonical-prefix G (length G) ! i
  from extract-eclass-global-minimum[OF wf i, of w]
  obtain chosen where
    extracted: extract-eclass w G i = Some chosen and
    represented: represents-eclass G i chosen and
    minimal: ⋀u. represents-eclass G i u ⟹
      term-cost w chosen ≤ term-cost w u
    by blast
  have source: canonical-eclass G i = Some ?source
    using wf i unfolding canonical-eclass-def by simp
  have stable:
    ?source = canonical-prefix G (Suc i) ! i
    by (rule canonical-prefix-nth-stable[OF i])
  have sound:
    (?source, chosen) ∈ (rstep (set R))^↔*
    unfolding stable
    by (rule checked-egraph-representation-sound[
          OF checked i represented])
  from that[OF source extracted sound represented minimal] show thesis .
```

qed

end

12 Global extraction over a finite e-class DAG

theory *Examples-Acyclic-EGraph*
  imports *Certified-Acyclic-EGraph*
begin

datatype *dag-sym* = *Dag-X* | *Dag-One* | *Dag-Two* | *Dag-Add* | *Dag-Mul* | *Dag-Shl*

fun *dag-cost* :: *dag-sym* ⇒ *nat* **where**
  dag-cost Dag-X = *0*
| *dag-cost Dag-One* = *1*
| *dag-cost Dag-Two* = *1*
| *dag-cost Dag-Add* = *1*
| *dag-cost Dag-Mul* = *4*
| *dag-cost Dag-Shl* = *5*

Classes 0–2 are leaves. Class 3 contains three representations of the same strength-reduced expression. Class 4 contains one binary addition e-node whose two children both refer to class 3. Consequently class 4 represents all nine combinations of the three child representations.

definition *strength-dag* :: *dag-sym acyclic-egraph* **where**
  strength-dag =
    [[(*Dag-X*, [])]
    , [(*Dag-One*, [])]
    , [(*Dag-Two*, [])]
    , [(*Dag-Shl*, [*0*, *1*]), (*Dag-Mul*, [*0*, *2*]), (*Dag-Add*, [*0*, *0*])]
    , [(*Dag-Add*, [*3*, *3*])]]

definition *dag-x* :: (*dag-sym*, *unit*) *term* **where**
  dag-x = *Fun Dag-X* []

definition *dag-best-child* :: (*dag-sym*, *unit*) *term* **where**
  dag-best-child = *Fun Dag-Add* [*dag-x*, *dag-x*]

definition *dag-best-root* :: (*dag-sym*, *unit*) *term* **where**
  dag-best-root = *Fun Dag-Add* [*dag-best-child*, *dag-best-child*]

definition *dag-canonical-child* :: (*dag-sym*, *unit*) *term* **where**
  dag-canonical-child = *Fun Dag-Shl* [*dag-x*, *Fun Dag-One* []]

definition *dag-canonical-root* :: (*dag-sym*, *unit*) *term* **where**
  dag-canonical-root =
    Fun Dag-Add [*dag-canonical-child*, *dag-canonical-child*]

definition *strength-dag-rules* :: (*dag-sym*, *unit*) *rule list* **where**
strength-dag-rules =
[(*dag-canonical-child*, *dag-best-child*)
, (*Fun Dag-Mul* [*dag-x*, *Fun Dag-Two* []], *dag-best-child*)]

definition *strength-dag-certificates* :: *dag-sym dag-certificates* **where**
strength-dag-certificates =
[[[]]
, [[]]
, [[]]
, [[]
, [*Rule-At* [] *0 Forward Var*, *Rule-At* [] *1 Backward Var*]
, [*Rule-At* [] *0 Forward Var*]]
, [[]]]

lemma *strength-dag-wf*:
wf-acyclic-egraph strength-dag
by *eval*

lemma *strength-dag-extracts*:
extract-eclass dag-cost strength-dag 4 = *Some dag-best-root*
by *eval*

lemma *strength-dag-certified*:
check-certified-egraph
strength-dag-rules strength-dag strength-dag-certificates
by *eval*

lemma *strength-dag-canonical*:
canonical-eclass strength-dag 4 = *Some dag-canonical-root*
by *eval*

definition *cyclic-dag* :: *dag-sym acyclic-egraph* **where**
cyclic-dag = [[(*Dag-Add*, [*0*, *0*])]]

lemma *cyclic-dag-not-wf*:
¬ *wf-acyclic-egraph cyclic-dag*
by *eval*

lemma *cyclic-dag-extraction-rejected*:
extract-egraph dag-cost cyclic-dag = *None*
unfolding *extract-egraph-def* **using** *cyclic-dag-not-wf* **by** *simp*

lemma *cyclic-dag-certificate-rejected*:
¬ *check-certified-egraph* [] *cyclic-dag* [[[]]]
unfolding *check-certified-egraph-def* **using** *cyclic-dag-not-wf* **by** *simp*

lemma *strength-dag-child-alternatives*:
represents-eclass strength-dag 3

```
    (Fun Dag-Shl [dag-x, Fun Dag-One []])
  represents-eclass strength-dag 3
    (Fun Dag-Mul [dag-x, Fun Dag-Two []])
  represents-eclass strength-dag 3 dag-best-child
  unfolding dag-x-def dag-best-child-def strength-dag-def
  by (auto intro!: represents-eclass.intros)

lemma strength-dag-root-has-cross-product:
  represents-eclass strength-dag 4
    (Fun Dag-Add
      [Fun Dag-Shl [dag-x, Fun Dag-One []],
       Fun Dag-Mul [dag-x, Fun Dag-Two []]])
  unfolding strength-dag-def dag-x-def
  by (auto intro!: represents-eclass.intros)

theorem strength-dag-global-minimum:
  assumes represents-eclass strength-dag 4 t
  shows term-cost dag-cost dag-best-root ≤ term-cost dag-cost t
proof −
  have index: 4 < length strength-dag by eval
  from extract-eclass-global-minimum[
      OF strength-dag-wf index, of dag-cost]
  obtain best where
    extracted: extract-eclass dag-cost strength-dag 4 = Some best and
    minimal: ⋀u. represents-eclass strength-dag 4 u ⟹
      term-cost dag-cost best ≤ term-cost dag-cost u
    by blast
  from extracted strength-dag-extracts have best = dag-best-root by simp
  with minimal[OF assms] show ?thesis by simp
qed

theorem strength-dag-all-represented-equal:
  assumes represents-eclass strength-dag 4 t
  shows (dag-canonical-root, t)
    ∈ (rstep (set strength-dag-rules))^↔*
proof −
  have index: 4 < length strength-dag by eval
  have (canonical-prefix strength-dag (Suc 4) ! 4, t)
      ∈ (rstep (set strength-dag-rules))^↔*
    by (rule checked-egraph-representation-sound[
          OF strength-dag-certified index assms])
  moreover have
    canonical-prefix strength-dag (Suc 4) ! 4 = dag-canonical-root
    by eval
  ultimately show ?thesis by simp
qed

theorem certified-strength-dag-extraction:
  (dag-canonical-root, dag-best-root)
```

```
    ∈ (rstep (set strength-dag-rules))^↔*
  represents-eclass strength-dag 4 dag-best-root
  ⋀u. represents-eclass strength-dag 4 u ⟹
    term-cost dag-cost dag-best-root ≤ term-cost dag-cost u
proof −
  have index: 4 < length strength-dag by eval
  from certified-egraph-extraction-correct[
      OF strength-dag-certified index, of dag-cost]
  obtain source chosen where
    source: canonical-eclass strength-dag 4 = Some source and
    chosen: extract-eclass dag-cost strength-dag 4 = Some chosen and
    sound:
      (source, chosen)
        ∈ (rstep (set strength-dag-rules))^↔* and
    represented: represents-eclass strength-dag 4 chosen and
    minimal: ⋀u. represents-eclass strength-dag 4 u ⟹
      term-cost dag-cost chosen ≤ term-cost dag-cost u
    by blast
  from source strength-dag-canonical have source-eq:
    source = dag-canonical-root by simp
  from chosen strength-dag-extracts have chosen-eq:
    chosen = dag-best-root by simp
  from sound source-eq chosen-eq show
    (dag-canonical-root, dag-best-root)
      ∈ (rstep (set strength-dag-rules))^↔*
    by simp
  from represented chosen-eq show
    represents-eclass strength-dag 4 dag-best-root by simp
  from minimal chosen-eq show
    term-cost dag-cost dag-best-root ≤ term-cost dag-cost u
    if represents-eclass strength-dag 4 u for u
    using that by simp
qed

value extract-egraph dag-cost strength-dag
value extract-eclass dag-cost strength-dag 4

end
```

13 A certificate-producing bounded search

```
theory Bounded-Certificate-Search
  imports Equality-Saturation-Checker
begin
```

The trusted result in this theory is independent of matching and scheduling. An arbitrary generator proposes certificate steps; only proposals accepted by *apply-step* enter the frontier. Consequently every path emitted by the bounded search is accepted by the independent checker.

definition *step-children* ::
  ('f, 'v) *rule list* ⇒ ('f, 'v) *rule list* ⇒
    (('f, 'v) *term* ⇒ ('f, 'v) *certificate-step list*) ⇒
    ('f, 'v) *term* ⇒ ('f, 'v) *certificate-step list* ⇒
    (('f, 'v) *term* × ('f, 'v) *certificate-step list*) *list* **where**
  step-children R Γ *gen u sts* =
     concat (*map* (λ*st*. *case apply-step R* Γ *st u of*
                        None ⇒ []
                      | *Some u'* ⇒ [(*u'*, *sts* @ [*st*])]) (*gen u*))

definition *expand* ::
  ('f, 'v) *rule list* ⇒ ('f, 'v) *rule list* ⇒
    (('f, 'v) *term* ⇒ ('f, 'v) *certificate-step list*) ⇒
    (('f, 'v) *term* × ('f, 'v) *certificate-step list*) ⇒
    (('f, 'v) *term* × ('f, 'v) *certificate-step list*) *list* **where**
  expand R Γ *gen p* = *step-children R* Γ *gen* (*fst p*) (*snd p*)

fun *iterate-search* ::
  ('f, 'v) *rule list* ⇒ ('f, 'v) *rule list* ⇒
    (('f, 'v) *term* ⇒ ('f, 'v) *certificate-step list*) ⇒ *nat* ⇒
    (('f, 'v) *term* × ('f, 'v) *certificate-step list*) *list* ⇒
    (('f, 'v) *term* × ('f, 'v) *certificate-step list*) *list* **where**
  iterate-search R Γ *gen 0 F* = *F*
| *iterate-search R* Γ *gen* (*Suc n*) *F* =
     iterate-search R Γ *gen n*
       (*F* @ *concat* (*map* (*expand R* Γ *gen*) *F*))

definition *run-bounded-search* ::
  ('f, 'v) *rule list* ⇒ ('f, 'v) *rule list* ⇒
    (('f, 'v) *term* ⇒ ('f, 'v) *certificate-step list*) ⇒ *nat* ⇒
    ('f, 'v) *term* ⇒
    (('f, 'v) *term* × ('f, 'v) *certificate-step list*) *list* **where**
  run-bounded-search R Γ *gen n t* =
    iterate-search R Γ *gen n* [(*t*, [])]

lemma *apply-steps-append*:
  apply-steps R Γ (*as* @ *bs*) *t* =
     (*case apply-steps R* Γ *as t of*
        None ⇒ *None*
      | *Some u* ⇒ *apply-steps R* Γ *bs u*)
  by (*induction as arbitrary*: *t*) (*auto split*: *option.splits*)

lemma *apply-steps-snoc*:
  assumes *apply-steps R* Γ *sts t* = *Some u*
    and *apply-step R* Γ *st u* = *Some u'*
  shows *apply-steps R* Γ (*sts* @ [*st*]) *t* = *Some u'*
  using *assms* **by** (*auto simp*: *apply-steps-append*)

lemma *step-children-accepted*:

```
  assumes apply-steps R Γ sts t = Some u
    and (u', sts') ∈ set (step-children R Γ gen u sts)
  shows apply-steps R Γ sts' t = Some u'
proof −
  from assms(2) obtain st where
    emit: (u', sts') ∈
      set (case apply-step R Γ st u of
            None ⇒ []
          | Some v ⇒ [(v, sts @ [st])])
    unfolding step-children-def by auto
  then obtain v where
    accepted: apply-step R Γ st u = Some v and
    pair: (u', sts') = (v, sts @ [st])
    by (auto split: option.splits)
  from apply-steps-snoc[OF assms(1) accepted] pair show ?thesis by simp
qed

lemma expand-accepted:
  assumes apply-steps R Γ (snd p) t = Some (fst p)
    and q ∈ set (expand R Γ gen p)
  shows apply-steps R Γ (snd q) t = Some (fst q)
proof −
  obtain u sts where q: q = (u, sts) by (cases q)
  from assms(2) q have
    (u, sts) ∈ set (step-children R Γ gen (fst p) (snd p))
    by (simp add: expand-def)
  from step-children-accepted[OF assms(1) this] q show ?thesis by simp
qed

lemma iterate-search-accepted:
  assumes ∀ p ∈ set F.
      apply-steps R Γ (snd p) t = Some (fst p)
  shows ∀ p ∈ set (iterate-search R Γ gen n F).
      apply-steps R Γ (snd p) t = Some (fst p)
  using assms
proof (induction n arbitrary: F)
  case 0
  then show ?case by simp
next
  case (Suc n)
  let ?F = F @ concat (map (expand R Γ gen) F)
  have ∀ p ∈ set ?F.
      apply-steps R Γ (snd p) t = Some (fst p)
  proof
    fix p
    assume p ∈ set ?F
    then consider p ∈ set F
      | p ∈ set (concat (map (expand R Γ gen) F))
      by auto
```

```
    then show apply-steps R Γ (snd p) t = Some (fst p)
    proof cases
      case 1
      then show ?thesis using Suc.prems by blast
    next
      case 2
      then obtain q where
        q ∈ set F and p ∈ set (expand R Γ gen q)
        by auto
      then show ?thesis using expand-accepted Suc.prems by blast
    qed
  qed
  from Suc.IH[OF this] show ?case by simp
qed

theorem run-bounded-search-accepted:
  assumes (u, sts) ∈ set (run-bounded-search R Γ gen n t)
  shows check-explanation R Γ sts t u
proof −
  have ∀ p ∈ set [(t, [])].
      apply-steps R Γ (snd p) t = Some (fst p)
    by simp
  from assms iterate-search-accepted[OF this]
  have apply-steps R Γ sts t = Some u
    by (fastforce simp: run-bounded-search-def)
  then show ?thesis by (simp add: check-explanation-def)
qed

corollary run-bounded-search-sound:
  assumes merges: ∀ ab ∈ set Γ.
      (fst ab, snd ab) ∈ (rstep (set R))^↔*
    and result: (u, sts) ∈ set (run-bounded-search R Γ gen n t)
  shows (t, u) ∈ (rstep (set R))^↔*
  by (rule check-explanation-sound[
        OF merges run-bounded-search-accepted[OF result]])

definition find-explanation ::
  ('f, 'v) rule list ⇒ ('f, 'v) rule list ⇒
    (('f, 'v) term ⇒ ('f, 'v) certificate-step list) ⇒ nat ⇒
    ('f, 'v) term ⇒ ('f, 'v) term ⇒
    ('f, 'v) certificate-step list option where
  find-explanation R Γ gen n t u =
     (case filter (λp. fst p = u)
       (run-bounded-search R Γ gen n t) of
       [] ⇒ None
     | p # - ⇒ Some (snd p))

theorem find-explanation-accepted:
  assumes find-explanation R Γ gen n t u = Some sts
```

```
  shows check-explanation R Γ sts t u
proof −
  from assms have
    filter (λp. fst p = u)
      (run-bounded-search R Γ gen n t) ≠ []
    unfolding find-explanation-def by (auto split: list.splits)
  then obtain p ps where
    filter: filter (λp. fst p = u)
      (run-bounded-search R Γ gen n t) = p # ps
    by (cases filter (λp. fst p = u)
          (run-bounded-search R Γ gen n t)) auto
  from assms filter have snd-p: snd p = sts
    unfolding find-explanation-def by simp
  have in-filter:
    p ∈ set (filter (λp. fst p = u)
      (run-bounded-search R Γ gen n t))
    using filter by simp
  then have mem:
    p ∈ set (run-bounded-search R Γ gen n t) by simp
  from in-filter have fst-p: fst p = u by simp
  from fst-p snd-p have p: p = (u, sts) by (cases p) simp
  from mem p have
    (u, sts) ∈ set (run-bounded-search R Γ gen n t) by simp
  then show ?thesis by (rule run-bounded-search-accepted)
qed

corollary find-explanation-sound:
  assumes ∀ ab ∈ set Γ.
      (fst ab, snd ab) ∈ (rstep (set R))^↔*
    and find-explanation R Γ gen n t u = Some sts
  shows (t, u) ∈ (rstep (set R))^↔*
  by (rule check-explanation-sound[OF assms(1)
        find-explanation-accepted[OF assms(2)]])

end
```

14 Compiler-style equality-saturation certificates

```
theory Examples-Compiler
  imports
    EGraph-Explanations
    Extraction-Certificates
    Bounded-Certificate-Search
begin

datatype csym = CMul | CAdd | CShl | COne | CTwo
datatype cvar = VX | VY

abbreviation (input) cmul ::
```

(*csym*, *cvar*) *term* ⇒ (*csym*, *cvar*) *term* ⇒ (*csym*, *cvar*) *term*
(**infixl** ‹⊗› *70*) **where**
a ⊗ *b* ≡ *Fun CMul* [*a*, *b*]

abbreviation (*input*) *cadd* ::
(*csym*, *cvar*) *term* ⇒ (*csym*, *cvar*) *term* ⇒ (*csym*, *cvar*) *term*
(**infixl** ‹⊕› *65*) **where**
a ⊕ *b* ≡ *Fun CAdd* [*a*, *b*]

abbreviation (*input*) *cshl* ::
(*csym*, *cvar*) *term* ⇒ (*csym*, *cvar*) *term* ⇒ (*csym*, *cvar*) *term*
(**infixl** ‹≪› *60*) **where**
a ≪ *b* ≡ *Fun CShl* [*a*, *b*]

abbreviation (*input*) *cone* :: (*csym*, *cvar*) *term* **where**
cone ≡ *Fun COne* []
abbreviation (*input*) *ctwo* :: (*csym*, *cvar*) *term* **where**
ctwo ≡ *Fun CTwo* []
abbreviation (*input*) *vx* :: (*csym*, *cvar*) *term* **where**
vx ≡ *Var VX*
abbreviation (*input*) *vy* :: (*csym*, *cvar*) *term* **where**
vy ≡ *Var VY*

definition *R-comp* :: (*csym*, *cvar*) *rule list* **where**
R-comp =
[(*vx* ⊗ *ctwo*, *vx* ⊕ *vx*)
, (*vx* ≪ *cone*, *vx* ⊕ *vx*)]

14.1 Positional rule explanations

definition *t-strength* :: (*csym*, *cvar*) *term* **where**
t-strength = (*vx* ≪ *cone*) ⊕ (*vy* ⊗ *ctwo*)

definition *u-strength* :: (*csym*, *cvar*) *term* **where**
u-strength = (*vx* ⊕ *vx*) ⊕ (*vy* ⊕ *vy*)

definition *strength-steps* :: (*csym*, *cvar*) *certificate-step list* **where**
strength-steps =
[*Rule-At* [*0*] *1 Forward Var*
, *Rule-At* [*Suc 0*] *0 Forward*
(λ*v*. *if v* = *VX then vy else Var v*)]

lemma *strength-certificate*:
check-explanation R-comp [] *strength-steps t-strength u-strength*
by (*simp add*: *check-explanation-def R-comp-def strength-steps-def*
t-strength-def u-strength-def)

theorem *strength-reduction-conversion*:
(*t-strength*, *u-strength*) ∈ (*rstep* (*set R-comp*))$^{\leftrightarrow *}$

by (*rule check-rule-explanation-sound*[*OF strength-certificate*])

14.2 Chronological merge reuse

definition *compiler-log* :: (*csym*, *cvar*) *merge-log* **where**
compiler-log =
[((*vx* ⊗ *ctwo*, *vx* ⊕ *vx*),
[*Rule-At* [] *0 Forward Var*])
, ((*vy* ⊗ *ctwo*, *vy* ⊕ *vy*),
[*Rule-At* [] *0 Forward*
(λ*v*. *if v* = *VX then vy else Var v*)])]

lemma *compiler-log-accepted*:
check-merge-log R-comp compiler-log
by (*simp add*: *check-merge-log-def compiler-log-def*
check-explanation-def R-comp-def)

definition *t-merge* :: (*csym*, *cvar*) *term* **where**
t-merge = (*vx* ⊗ *ctwo*) ⊕ (*vy* ⊗ *ctwo*)

definition *u-merge* :: (*csym*, *cvar*) *term* **where**
u-merge = (*vx* ⊕ *vx*) ⊕ (*vy* ⊕ *vy*)

definition *merge-reuse-steps* :: (*csym*, *cvar*) *certificate-step list* **where**
merge-reuse-steps =
[*Merge-At* [*0*] *0 Forward*
, *Merge-At* [*Suc 0*] *1 Forward*]

lemma *merge-reuse-accepted*:
check-explanation R-comp (*recorded-merges compiler-log*)
merge-reuse-steps t-merge u-merge
by (*simp add*: *check-explanation-def recorded-merges-def compiler-log-def*
merge-reuse-steps-def t-merge-def u-merge-def)

theorem *recorded-merges-are-reusable*:
(*t-merge*, *u-merge*) ∈ (*rstep* (*set R-comp*))$^{\leftrightarrow *}$
by (*rule egraph-explanation-sound*[
OF compiler-log-accepted merge-reuse-accepted])

14.3 Candidate-set extraction

fun *compiler-cost* :: *csym* ⇒ *nat* **where**
compiler-cost CMul = *4*
| *compiler-cost CShl* = *4*
| *compiler-cost CAdd* = *2*
| *compiler-cost COne* = *1*
| *compiler-cost CTwo* = *1*

definition *compiler-extraction* :: (*csym*, *cvar*) *extraction* **where**
compiler-extraction =

```
    (| ex-source = vx ⊗ ctwo
    , ex-chosen = vx ⊕ vx
    , ex-candidates =
        [ (vx ⊕ vx, [Merge-At [] 0 Forward])
        , (vx ⊗ ctwo, []) ] |)

lemma compiler-extraction-accepted:
  check-extraction compiler-cost R-comp
    (recorded-merges compiler-log) compiler-extraction
  by (simp add: check-extraction-def check-explanation-def
      recorded-merges-def compiler-log-def compiler-extraction-def)

theorem compiler-extraction-correct:
  (ex-source compiler-extraction, ex-chosen compiler-extraction)
    ∈ (rstep (set R-comp))^{↔*}
  ∀ v ∈ set (map fst (ex-candidates compiler-extraction)).
    term-cost compiler-cost (ex-chosen compiler-extraction)
      ≤ term-cost compiler-cost v
  using extraction-over-checked-log-correct[
    OF compiler-log-accepted compiler-extraction-accepted] .
```

14.4 Certificate production with AFP position enumeration

```
definition forward-generator ::
  ('f, 'v) rule list ⇒ ('f, 'v) term ⇒
    ('f, 'v) certificate-step list where
  forward-generator R t =
    [Rule-At p i Forward Var.
      p ← poss-list t, i ← [0 ..< length R]]

definition zero :: (string, string) term where
  zero = Fun ''0'' []
definition aval :: (string, string) term where
  aval = Fun ''a'' []
definition bval :: (string, string) term where
  bval = Fun ''b'' []
definition plus-term ::
  (string, string) term ⇒ (string, string) term ⇒
    (string, string) term where
  plus-term s t = Fun ''+'' [s, t]
definition times-term ::
  (string, string) term ⇒ (string, string) term ⇒
    (string, string) term where
  times-term s t = Fun ''*'' [s, t]

definition demo-rules :: (string, string) rule list where
  demo-rules =
    [(plus-term aval zero, aval), (plus-term bval zero, bval)]
definition demo-start :: (string, string) term where
```

```
  demo-start = times-term (plus-term aval zero) (plus-term bval zero)
definition demo-result :: (string, string) term where
  demo-result = times-term aval bval

lemma demo-search-succeeds:
  find-explanation demo-rules [] (forward-generator demo-rules) 2
    demo-start demo-result ≠ None
  by eval

theorem produced-certificate-is-sound:
  (demo-start, demo-result) ∈ (rstep (set demo-rules))^↔*
proof −
  from demo-search-succeeds obtain sts where found:
    find-explanation demo-rules [] (forward-generator demo-rules) 2
      demo-start demo-result = Some sts
    by blast
  show ?thesis by (rule find-explanation-sound[OF - found]) simp
qed

end
```

15 End-to-end fixtures emitted by egg 0.11.0

```
theory Examples-Egg
  imports Egg-Flat-Explanation
begin
```

The first two traces below are literal elements returned by egg 0.11.0's flat-string explanation API. They were generated deterministically with explanation-length optimization disabled. Egg's own proof checker validated both traces. The pinned package checksum and generator setup are recorded in the AFP document and README. The generator used the four named pattern rules below in the order given by `egg_compiler_rules`.

```
definition egg-compiler-rules :: egg-named-rules where
  egg-compiler-rules =
    [ (''mul−two'',
       (Fun ''∗'' [Var ''x'', Fun ''2'' []],
        Fun ''+'' [Var ''x'', Var ''x'']))
    , (''shl−one'',
       (Fun ''shl'' [Var ''x'', Fun ''1'' []],
        Fun ''+'' [Var ''x'', Var ''x'']))
    , (''mul−one'',
       (Fun ''∗'' [Var ''x'', Fun ''1'' []],
        Var ''x''))
    , (''add−zero'',
       (Fun ''+'' [Var ''x'', Fun ''0'' []],
        Var ''x'')) ]
```

definition *egg-compiler-start* :: (*string*, *string*) *term* **where**
egg-compiler-start =
Fun ''+''
[*Fun* ''*shl*'' [*Fun* ''*x*'' [], *Fun* ''*1*'' []],
Fun ''∗'' [*Fun* ''*y*'' [], *Fun* ''*2*'' []]]

definition *egg-compiler-final* :: (*string*, *string*) *term* **where**
egg-compiler-final =
Fun ''+''
[*Fun* ''+'' [*Fun* ''*x*'' [], *Fun* ''*x*'' []],
Fun ''+'' [*Fun* ''*y*'' [], *Fun* ''*y*'' []]]

definition *egg-compiler-flat-output* :: *string list* **where**
egg-compiler-flat-output =
[''(+ (*shl x 1*) (∗ *y 2*))'',
''(+ (*Rewrite=> shl−one* (+ *x x*)) (∗ *y 2*))'',
''(+ (+ *x x*) (*Rewrite=> mul−two* (+ *y y*)))'']

lemma *egg-compiler-fixture-accepted*:
check-egg-explanation egg-compiler-rules egg-compiler-flat-output
egg-compiler-start egg-compiler-final
by *eval*

theorem *egg-compiler-explanation-sound*:
(*egg-compiler-start*, *egg-compiler-final*)
∈ (*rstep* (*set* (*map snd egg-compiler-rules*)))$^{\leftrightarrow *}$
by (*rule check-egg-explanation-sound*[*OF egg-compiler-fixture-accepted*])

The second fixture reverses the query order after egg has applied *mul−one*. It therefore exercises egg's actual *Rewrite<=* convention: applying *mul−one* left-to-right to the current annotated term recovers the previous term.

definition *egg-backward-start* :: (*string*, *string*) *term* **where**
egg-backward-start = *Fun* ''*z*'' []

definition *egg-backward-final* :: (*string*, *string*) *term* **where**
egg-backward-final = *Fun* ''∗'' [*Fun* ''*z*'' [], *Fun* ''*1*'' []]

definition *egg-backward-flat-output* :: *string list* **where**
egg-backward-flat-output =
[''*z*'', ''(*Rewrite<= mul−one* (∗ *z 1*))'']

lemma *egg-backward-fixture-accepted*:
check-egg-explanation egg-compiler-rules egg-backward-flat-output
egg-backward-start egg-backward-final
by *eval*

theorem *egg-backward-explanation-sound*:
(*egg-backward-start*, *egg-backward-final*)
∈ (*rstep* (*set* (*map snd egg-compiler-rules*)))$^{\leftrightarrow *}$

by (*rule check-egg-explanation-sound*[*OF egg-backward-fixture-accepted*])

The third egg trace is a parser regression fixture. Its enclosing binary node has an atomic first child and an annotated second child, so the rewrite must be located at AFP position [*1*].

definition *egg-atomic-child-start* :: (*string*, *string*) *term* **where**
egg-atomic-child-start =
Fun ''*f*''
[*Fun* ''*y*'' [],
Fun ''+'' [*Fun* ''*z*'' [], *Fun* ''*0*'' []]]

definition *egg-atomic-child-final* :: (*string*, *string*) *term* **where**
egg-atomic-child-final = *Fun* ''*f*'' [*Fun* ''*y*'' [], *Fun* ''*z*'' []]

definition *egg-atomic-child-flat-output* :: *string list* **where**
egg-atomic-child-flat-output =
[''(*f y* (+ *z 0*))'', ''(*f y* (*Rewrite=> add−zero z*))'']

lemma *egg-atomic-child-fixture-accepted*:
check-egg-explanation egg-compiler-rules egg-atomic-child-flat-output
egg-atomic-child-start egg-atomic-child-final
by *eval*

theorem *egg-atomic-child-explanation-sound*:
(*egg-atomic-child-start*, *egg-atomic-child-final*)
∈ (*rstep* (*set* (*map snd egg-compiler-rules*)))$^{\leftrightarrow *}$
by (*rule check-egg-explanation-sound*[
OF egg-atomic-child-fixture-accepted])

15.1 Rejecting malformed or unsupported traces

lemma *egg-multiple-annotations-rejected*:
decode-egg-line
''(+ (*Rewrite=> mul−one x*) (*Rewrite=> mul−one y*))'' = *None*
by *eval*

lemma *egg-unknown-rule-rejected*:
¬ *check-egg-explanation egg-compiler-rules*
[''*x*'', ''(*Rewrite=> unknown y*)'']
(*Fun* ''*x*'' []) (*Fun* ''*y*'' [])
by *eval*

end

16 Executable checker and producer

theory *Code-Generation*
imports *Examples-Compiler Examples-Egg Examples-Acyclic-EGraph*

begin

Only the equality-saturation-specific layer is exported. Terms, substitutions, positions, and position enumeration come from `First_Order_Terms`; the specification theorem targets `First_Order_Rewriting`. The exported acyclic-e-graph functions include both the global dynamic-programming extractor and the certificate checker for e-class equality.

export-code
apply-step apply-steps check-explanation
tokenize-egg parse-egg-sexp decode-egg-line
compile-egg-explanation check-egg-explanation
check-merge-log-from check-merge-log recorded-merges
check-extraction
wf-acyclic-egraph instantiate-enode best-eclass-term
extract-prefix extract-egraph extract-eclass
canonical-prefix canonical-eclass check-dag-class check-certified-egraph
run-bounded-search find-explanation
in *SML*

export-code
apply-step apply-steps check-explanation
tokenize-egg parse-egg-sexp decode-egg-line
compile-egg-explanation check-egg-explanation
check-merge-log-from check-merge-log recorded-merges
check-extraction
wf-acyclic-egraph instantiate-enode best-eclass-term
extract-prefix extract-egraph extract-eclass
canonical-prefix canonical-eclass check-dag-class check-certified-egraph
run-bounded-search find-explanation
in *Haskell* **module-name** *Equality-Saturation-Checker*

definition *produced-steps* :: (*string*, *string*) *certificate-step list* **where**
produced-steps =
the (*find-explanation demo-rules* [] (*forward-generator demo-rules*) *2*
demo-start demo-result)

lemma *produced-steps-accepted*:
check-explanation demo-rules [] *produced-steps demo-start demo-result*
proof −
have *find-explanation demo-rules* [] (*forward-generator demo-rules*) *2*
demo-start demo-result = *Some produced-steps*
unfolding *produced-steps-def*
using *demo-search-succeeds*
by (*cases find-explanation demo-rules* []
(*forward-generator demo-rules*) *2 demo-start demo-result*) *auto*
then show *?thesis* **by** (*rule find-explanation-accepted*)
qed

value *check-explanation demo-rules* [] *produced-steps demo-start demo-result*

value *check-egg-explanation egg-compiler-rules egg-compiler-flat-output*
egg-compiler-start egg-compiler-final
value *map* (*λst. case st of*
Rule-At p i d σ ⇒ (*p*, *i*, *d*)
| *Merge-At p i d ⇒* (*p*, *i*, *d*)) *produced-steps*

end

References

[1] egg contributors. egg 0.11.0 explanation api. Rust crate documentation and source, 2025. https://docs.rs/egg/0.11.0/egg/struct.Explanation.html; source tag https://github.com/egraphs-good/egg/tree/v0.11.0.

[2] C. Kirk. Proof Terms for Term Rewriting. Archive of Formal Proofs, 2025. https://isa-afp.org/entries/Proof_Terms_Term_Rewriting.html.

[3] C. Sternagel and R. Thiemann. Equational Reasoning. IsaFoR theory source, 2018. Theory source: http://cl2-informatik.uibk.ac.at/rewriting/mercurial.cgi/IsaFoR/file/c7b7a9d1b6e8/thys/Confluence_and_Completion/Equational_Reasoning.thy; project site: https://isafor-ceta.uibk.ac.at/.

[4] C. Sternagel and R. Thiemann. First-Order Terms. Archive of Formal Proofs, 2018. https://isa-afp.org/entries/First_Order_Terms.html.

[5] C. Sternagel and S. Winkler. Certified Equational Reasoning via Ordered Completion. In *Automated Deduction – CADE 27*, volume 11716 of *Lecture Notes in Computer Science*, pages 508–525. Springer, 2019. DOI: https://doi.org/10.1007/978-3-030-29436-6_30.

[6] T. Sternagel, S. Winkler, and H. Zankl. Recording Completion for Certificates in Equational Reasoning. In *Proceedings of the 4th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 41–47. ACM, 2015. DOI: https://doi.org/10.1145/2676724.2693171.

[7] R. Thiemann and C. Sternagel. Certification of Termination Proofs using CeTA. In *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009. DOI: https://doi.org/10.1007/978-3-642-03359-9_31.

[8] R. Thiemann, C. Sternagel, C. Kirk, M. Avanzini, B. Felgenhauer, J. Nagele, T. Sternagel, S. Winkler, and A. Yamada. First-Order Rewriting. Archive of Formal Proofs, 2025. https://isa-afp.org/entries/First_Order_Rewriting.html.

[9] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021. DOI: https://doi.org/10.1145/3434304.