# Checking Equality-Saturation Merge and Extraction Certificates

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

July 16, 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⟩

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⟩

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⟩
```

**lemma** *apply-steps-sound*:
**assumes** *merges*:
$\forall\, ab \in set\ \Gamma.\ (fst\ ab,\ snd\ ab) \in (rstep\ (set\ R))^{\leftrightarrow *}$
**and** *run*: *apply-steps* $R\ \Gamma\ sts\ t = Some\ u$
**shows** $(t,\ u) \in (rstep\ (set\ R))^{\leftrightarrow *}$
⟨*proof*⟩

**theorem** *check-explanation-sound*:
**assumes** $\forall\, ab \in set\ \Gamma.$
$(fst\ ab,\ snd\ ab) \in (rstep\ (set\ R))^{\leftrightarrow *}$
**and** *check-explanation* $R\ \Gamma\ sts\ t\ u$
**shows** $(t,\ u) \in (rstep\ (set\ R))^{\leftrightarrow *}$
⟨*proof*⟩

**corollary** *check-rule-explanation-sound*:
**assumes** *check-explanation* $R\ []\ sts\ t\ u$
**shows** $(t,\ u) \in (rstep\ (set\ R))^{\leftrightarrow *}$
⟨*proof*⟩

**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⟩

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* $\times$ $('f, 'v)$ *certificate-step list*$)$ *list*

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

**definition** *check-merge-log* ::
  $('f, 'v)$ *rule list* $\Rightarrow$ $('f, 'v)$ *merge-log* $\Rightarrow$ *bool* **where**
  *check-merge-log R log* $\longleftrightarrow$ *check-merge-log-from R* [] *log*

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

**lemma** *check-merge-log-from-sound*:
  **assumes** *check*: *check-merge-log-from R* $\Gamma$ *es*
    **and** *base*: $\forall$ *ab* $\in$ *set* $\Gamma$.
      (*fst ab*, *snd ab*) $\in$ (*rstep* (*set R*))$^{\leftrightarrow *}$
  **shows** $\forall$ *ab* $\in$ *set* ($\Gamma$ @ *map fst es*).
      (*fst ab*, *snd ab*) $\in$ (*rstep* (*set R*))$^{\leftrightarrow *}$
  ⟨*proof*⟩

**theorem** *checked-merge-log-sound*:
  **assumes** *check-merge-log R log*
    **and** *ab* $\in$ *set* (*recorded-merges log*)
  **shows** (*fst ab*, *snd ab*) $\in$ (*rstep* (*set R*))$^{\leftrightarrow *}$
⟨*proof*⟩

**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*))$^{\leftrightarrow *}$
⟨*proof*⟩

**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*))$^{\leftrightarrow *}$
    **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*))$^{\leftrightarrow *}$
⟨*proof*⟩

**lemma** *check-extraction-chosen-sound*:

**assumes** $\forall\, ab \in set\ \Gamma.$
$(fst\ ab,\ snd\ ab) \in (rstep\ (set\ R))^{\leftrightarrow *}$
**and** *check-extraction w R* $\Gamma$ *E*
**shows** $(ex\text{-}source\ E,\ ex\text{-}chosen\ E) \in (rstep\ (set\ R))^{\leftrightarrow *}$
⟨*proof*⟩

**lemma** *check-extraction-minimal*:
**assumes** *check-extraction w R* $\Gamma$ *E*
**and** $v \in set\ (map\ fst\ (ex\text{-}candidates\ E))$
**shows** *term-cost w* $(ex\text{-}chosen\ E) \leq$ *term-cost w v*
⟨*proof*⟩

**theorem** *extraction-over-checked-log-correct*:
**assumes** *log*: *check-merge-log R mlog*
**and** *extraction*:
*check-extraction w R* (*recorded-merges mlog*) *E*
**shows** $(ex\text{-}source\ E,\ ex\text{-}chosen\ E)$
$\in (rstep\ (set\ R))^{\leftrightarrow *}$
**and** $\forall\, v \in set\ (map\ fst\ (ex\text{-}candidates\ E)).$
*term-cost w* $(ex\text{-}chosen\ E) \leq$ *term-cost w v*
⟨*proof*⟩

**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*
⟨*proof*⟩

**lemma** *extract-prefix-nth-stable*:
**assumes** *j* < *n*
**shows** *extract-prefix w G n* ! *j* = *extract-prefix w G* (*Suc j*) ! *j*
⟨*proof*⟩

**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*⟩

**lemma** *sum-list-mono-list-all2*:
**fixes** *xs ys* :: *nat list*
**assumes** *list-all2* (≤) *xs ys*
**shows** *sum-list xs* ≤ *sum-list ys*
⟨*proof*⟩

**lemma** *finite-list-all2-fibres*:
**assumes** ∀ *x* ∈ *set xs*. *finite* {*y*. *P x y*}
**shows** *finite* {*ys*. *list-all2 P xs ys*}
⟨*proof*⟩

**theorem** *represented-eclass-finite*:
**assumes** *wf*: *wf-acyclic-egraph G* **and** *i*: *i* < *length G*
**shows** *finite* {*t*. *represents-eclass G i t*}
⟨*proof*⟩

**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*
⟨*proof*⟩

**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*⟩

**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*⟩

**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*
(λ*node 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*
(λ*i*. *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*
⟨*proof*⟩

**lemma** *canonical-prefix-nth-stable*:
**assumes** *j* < *n*
**shows** *canonical-prefix G n* ! *j* = *canonical-prefix G* (*Suc j*) ! *j*
⟨*proof*⟩

**lemma** *canonical-prefix-step*:
*canonical-prefix G* (*Suc i*) ! *i* =
*instantiate-enode* (*canonical-prefix G i*) (*hd* (*G* ! *i*))
⟨*proof*⟩

**lemma** *checked-egraph-wf*:
*check-certified-egraph R G certs* ⟹ *wf-acyclic-egraph G*
⟨*proof*⟩

**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*)
⟨*proof*⟩

**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*⟩

**lemma** *conversion-Fun-list-all2*:
**assumes**
*list-all2*
(λ*s t*. (*s*, *t*) ∈ (*rstep R*)$^{\leftrightarrow *}$) *ss ts*
**shows** (*Fun f ss*, *Fun f ts*) ∈ (*rstep R*)$^{\leftrightarrow *}$
⟨*proof*⟩

**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*)
∈ (*rstep* (*set R*))$^{\leftrightarrow *}$
⟨*proof*⟩

**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*))$^{\leftrightarrow *}$
⟨*proof*⟩

**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*))$^{\leftrightarrow *}$
*represents-eclass G i chosen*
$\bigwedge u$. *represents-eclass G i u* ⟹
*term-cost w chosen* ≤ *term-cost w u*

⟨*proof*⟩

**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*
⟨*proof*⟩

**lemma** *strength-dag-extracts*:
*extract-eclass dag-cost strength-dag 4* = *Some dag-best-root*
⟨*proof*⟩

**lemma** *strength-dag-certified*:
*check-certified-egraph*
*strength-dag-rules strength-dag strength-dag-certificates*
⟨*proof*⟩

**lemma** *strength-dag-canonical*:
*canonical-eclass strength-dag 4* = *Some dag-canonical-root*
⟨*proof*⟩

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

**lemma** *cyclic-dag-not-wf*:
¬ *wf-acyclic-egraph cyclic-dag*
⟨*proof*⟩

**lemma** *cyclic-dag-extraction-rejected*:
*extract-egraph dag-cost cyclic-dag* = *None*
⟨*proof*⟩

**lemma** *cyclic-dag-certificate-rejected*:
¬ *check-certified-egraph* [] *cyclic-dag* [[[]]]
⟨*proof*⟩

**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*
⟨*proof*⟩

**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* []]])
⟨*proof*⟩

**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*⟩

**theorem** *strength-dag-all-represented-equal*:
**assumes** *represents-eclass strength-dag 4 t*
**shows** (*dag-canonical-root*, *t*)
∈ (*rstep* (*set strength-dag-rules*))$^{\leftrightarrow *}$
⟨*proof*⟩

**theorem** *certified-strength-dag-extraction*:
(*dag-canonical-root*, *dag-best-root*)
∈ (*rstep* (*set strength-dag-rules*))$^{\leftrightarrow *}$
*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*⟩

**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 (λst. 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)
  ⟨proof⟩

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'
  ⟨proof⟩

lemma step-children-accepted:
  assumes apply-steps R Γ sts t = Some u
```

**and** $(u', sts') \in set\ (step\text{-}children\ R\ \Gamma\ gen\ u\ sts)$
**shows** $apply\text{-}steps\ R\ \Gamma\ sts'\ t = Some\ u'$
⟨*proof*⟩

**lemma** *expand-accepted*:
**assumes** $apply\text{-}steps\ R\ \Gamma\ (snd\ p)\ t = Some\ (fst\ p)$
**and** $q \in set\ (expand\ R\ \Gamma\ gen\ p)$
**shows** $apply\text{-}steps\ R\ \Gamma\ (snd\ q)\ t = Some\ (fst\ q)$
⟨*proof*⟩

**lemma** *iterate-search-accepted*:
**assumes** $\forall\, p \in set\ F.$
$apply\text{-}steps\ R\ \Gamma\ (snd\ p)\ t = Some\ (fst\ p)$
**shows** $\forall\, p \in set\ (iterate\text{-}search\ R\ \Gamma\ gen\ n\ F).$
$apply\text{-}steps\ R\ \Gamma\ (snd\ p)\ t = Some\ (fst\ p)$
⟨*proof*⟩

**theorem** *run-bounded-search-accepted*:
**assumes** $(u, sts) \in set\ (run\text{-}bounded\text{-}search\ R\ \Gamma\ gen\ n\ t)$
**shows** $check\text{-}explanation\ R\ \Gamma\ sts\ t\ u$
⟨*proof*⟩

**corollary** *run-bounded-search-sound*:
**assumes** *merges*: $\forall\, ab \in set\ \Gamma.$
$(fst\ ab, snd\ ab) \in (rstep\ (set\ R))^{\leftrightarrow *}$
**and** *result*: $(u, sts) \in set\ (run\text{-}bounded\text{-}search\ R\ \Gamma\ gen\ n\ t)$
**shows** $(t, u) \in (rstep\ (set\ R))^{\leftrightarrow *}$
⟨*proof*⟩

**definition** *find-explanation* ::
$('f, 'v)\ rule\ list \Rightarrow ('f, 'v)\ rule\ list \Rightarrow$
$(('f, 'v)\ term \Rightarrow ('f, 'v)\ certificate\text{-}step\ list) \Rightarrow nat \Rightarrow$
$('f, 'v)\ term \Rightarrow ('f, 'v)\ term \Rightarrow$
$('f, 'v)\ certificate\text{-}step\ list\ option$ **where**
*find-explanation* $R\ \Gamma\ gen\ n\ t\ u =$
$(case\ filter\ (\lambda p.\ fst\ p = u)$
$(run\text{-}bounded\text{-}search\ R\ \Gamma\ gen\ n\ t)\ of$
$[] \Rightarrow None$
$\mid p\ \#\ \text{-} \Rightarrow Some\ (snd\ p))$

**theorem** *find-explanation-accepted*:
**assumes** $find\text{-}explanation\ R\ \Gamma\ gen\ n\ t\ u = Some\ sts$
**shows** $check\text{-}explanation\ R\ \Gamma\ sts\ t\ u$
⟨*proof*⟩

**corollary** *find-explanation-sound*:
**assumes** $\forall\, ab \in set\ \Gamma.$
$(fst\ ab, snd\ ab) \in (rstep\ (set\ R))^{\leftrightarrow *}$
**and** $find\text{-}explanation\ R\ \Gamma\ gen\ n\ t\ u = Some\ sts$

**shows** $(t,\ u) \in (rstep\ (set\ R))^{\leftrightarrow *}$
⟨*proof*⟩

**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**
$a \otimes b \equiv Fun\ CMul\ [a,\ b]$

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

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

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

**definition** *R-comp* :: (*csym*, *cvar*) *rule list* **where**
*R-comp* =
[ $(vx \otimes ctwo,\ vx \oplus vx)$
, $(vx \ll cone,\ vx \oplus vx)$ ]

## 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*
⟨*proof*⟩

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

## 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*
⟨*proof*⟩

**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*
⟨*proof*⟩

**theorem** *recorded-merges-are-reusable*:
(*t-merge*, *u-merge*) ∈ (*rstep* (*set R-comp*))$^{\leftrightarrow *}$
⟨*proof*⟩

## 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*
⟨*proof*⟩

**theorem** *compiler-extraction-correct*:
(*ex-source compiler-extraction*, *ex-chosen compiler-extraction*)
∈ (*rstep* (*set R-comp*))$^{\leftrightarrow *}$
∀ *v* ∈ *set* (*map fst* (*ex-candidates compiler-extraction*)).
*term-cost compiler-cost* (*ex-chosen compiler-extraction*)
≤ *term-cost compiler-cost v*
⟨*proof*⟩

## 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
  ⟨proof⟩

theorem produced-certificate-is-sound:
  (demo-start, demo-result) ∈ (rstep (set demo-rules))^↔*
⟨proof⟩

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
  ⟨proof⟩

theorem egg-compiler-explanation-sound:
  (egg-compiler-start, egg-compiler-final)
    ∈ (rstep (set (map snd egg-compiler-rules)))^↔*
  ⟨proof⟩
```

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

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

## 15.1 Rejecting malformed or unsupported traces

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

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

**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*⟩

**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* (λ*st*. *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.