

# Veriopt Theories

February 23, 2022

## Contents

**theory** *TreeSnippets*  
**imports**  
    *Canonicalizations.ConditionalPhase*  
    *Optimizations.CanonicalizationSyntax*  
    *Semantics.TreeToGraphThms*  
    *Snippets.Snipping*  
    *HOL-Library.OptionalSugar*  
**begin**  
  
**no-notation** *ConditionalExpr* (- ? - : -)  
  
**notation** (*latex*)  
    *kind* (-⟨-⟩)  
  
**notation** (*latex*)  
    *valid-value* (- ∈ -)  
  
**notation** (*latex*)  
    *val-to-bool* (*bool-of* -)  
  
**notation** (*latex*)  
    *constantAsStamp* (*stamp-from-value* -)  
  
**notation** (*latex*)  
    *size* (*trm*(-))  
  
**translations**  
     $y > x \leq x < y$   
  
**notation** (*latex*)  
    *greater* (- >/ -)

**translations**

$n \leq \text{CONST Rep-intexp } n$   
 $n \leq \text{CONST Rep-i32exp } n$   
 $n \leq \text{CONST Rep-i64exp } n$

**lemma** *vminusv*:  $\forall vv \ v. \ vv = \text{IntVal32 } v \longrightarrow v - v = 0$   
**by** *simp*  
**thm-oracles** *vminusv*

**lemma** *vminusv2*:  $\forall v::\text{int32}. \ v - v = 0$   
**by** *simp*

**lemma** *redundant-sub*:  
 $\forall vv_1 \ vv_2 \ v_1 \ v_2. \ vv_1 = \text{IntVal32 } v_1 \wedge vv_2 = \text{IntVal32 } v_2 \longrightarrow v_1 - (v_1 - v_2) = v_2$   
**by** *simp*  
**thm-oracles** *redundant-sub*

**lemma** *redundant-sub2*:  
 $\forall (v_1::\text{int32}) (v_2::\text{int32}). \ v_1 - (v_1 - v_2) = v_2$   
**by** *simp*

*val-eq*

$\forall vv \ v. \ vv = \text{IntVal32 } v \longrightarrow v - v = 0$

$\forall vv_1 \ vv_2 \ v_1 \ v_2. \ vv_1 = \text{IntVal32 } v_1 \wedge vv_2 = \text{IntVal32 } v_2 \longrightarrow v_1 - (v_1 - v_2) = v_2$

**phase** *tmp*  
**terminating** *size*  
**begin**

*sub-same-32*

**optimization** *sub-same-32*:  $(e::\text{i32exp}) - e \mapsto \text{const } (\text{IntVal32 } 0)$

**apply** (*unfold rewrite-preservation.simps*, *unfold rewrite-termination.simps*,  
*rule conjE*, *simp*) **apply** *auto*[1] **using** *Rep-i32exp evalDet is-IntVal32-def*  
**apply** (*smt (verit, del-insts) eq-iff-diff-eq-0 evaltree.simps int-constants-valid int-val-sub.simps(1) is-int-val.simps(1) mem-Collect-eq*)  
**unfolding** *size.simps*  
**by** (*metis add-strict-increasing gr-implies-not0 less-one linorder-not-le size-gt-0*)

*sub-same-64*

**optimization** *sub-same-64*:  $(e::\text{i64exp}) - e \mapsto \text{const } (\text{IntVal64 } 0)$

**apply** *auto*  
**apply** (*metis (no-types, opaque-lifting) ConstantExpr bin-eval.simps(3) bin-eval-preserves-validity cancel-comm-monoid-add-class.diff-cancel evalDet i64e-eval int-and-equal-bits.simps(2)*)

```

intval-sub.simps(2))
  by (simp add: Suc-le-eq add-strict-increasing size-gt-0)
end

```

**thm-oracles** *sub-same-32*

*ast-example*

```

BinaryExpr BinAdd (BinaryExpr BinMul x x) (BinaryExpr BinMul x x)

```

*abstract-syntax-tree*

```

datatype IRExpr =
  UnaryExpr IRUnaryOp IRExpr
| BinaryExpr IRBinaryOp IRExpr IRExpr
| ConditionalExpr IRExpr IRExpr IRExpr
| ParameterExpr nat Stamp
| LeafExpr nat Stamp
| ConstantExpr Value
| ConstantVar (char list)
| VariableExpr (char list) Stamp

```

*value*

```

datatype Value = UndefVal
| IntVal32 (32 word)
| IntVal64 (64 word)
| ObjRef (nat option)
| ObjStr (char list)

```

*eval*

```

unary-eval :: IRUnaryOp ⇒ Value ⇒ Value

bin-eval :: IRBinaryOp ⇒ Value ⇒ Value ⇒ Value

```

*tree-semantics*

```

semantics:unary  semantics:binary  semantics:conditional  seman-
tics:constant semantics:parameter semantics:leaf

```

*tree-evaluation-deterministic*

$$[m, p] \vdash e \mapsto v_1 \wedge [m, p] \vdash e \mapsto v_2 \implies v_1 = v_2$$

**thm-oracles** *evalDet*

*expression-refinement*

$$e_1 \sqsubseteq e_2 = (\forall m\ p\ v. [m, p] \vdash e_1 \mapsto v \longrightarrow [m, p] \vdash e_2 \mapsto v)$$

*expression-refinement-monotone*

$$\begin{aligned} e \sqsubseteq e' &\implies \text{UnaryExpr op } e \sqsubseteq \text{UnaryExpr op } e' \\ x \sqsubseteq x' \wedge y \sqsubseteq y' &\implies \text{BinaryExpr op } x\ y \sqsubseteq \text{BinaryExpr op } x'\ y' \\ ce \sqsubseteq ce' \wedge te \sqsubseteq te' \wedge fe \sqsubseteq fe' &\implies \text{ConditionalExpr } ce\ te\ fe \sqsubseteq \text{ConditionalExpr } ce'\ te'\ fe' \end{aligned}$$

**ML**  $\langle$

```
(*fun get-list (phase: phase option) =
  case phase of
    NONE => [] |
    SOME p => (#rewrites p)
```

```
fun get-rewrite name thy =
  let
    val (phases, lookup) = (case RWList.get thy of
      NoPhase store => store |
      InPhase (name, store, -) => store)
    val rewrites = (map (fn x => get-list (lookup x)) phases)
  in
    rewrites
  end
```

```
fun rule-print name =
  Document-Output.antiquotation-pretty name (Args.term)
  (fn ctxt => fn (rule) => (*Pretty.str hello*)
    Pretty.block (print-all-phases (Proof-Context.theory-of ctxt)));
(*
  Goal-Display.pretty-goal
  (Config.put Goal-Display.show-main-goal main ctxt)
  (#goal (Proof.goal (Toplevel.proof-of (Toplevel.presentation-state ctxt))));
*)
```

```
val - = Theory.setup
  (rule-print binding⟨rule⟩;*)
⟩
```

phase *SnipPhase*  
 terminating *size*  
 begin

*BinaryFoldConstant*

**optimization** *BinaryFoldConstant*:  $\text{BinaryExpr } op \text{ (const } v1 \text{) (const } v2 \text{)} \mapsto \text{ConstantExpr (bin-eval } op \text{ } v1 \text{ } v2 \text{)}$  when *int-and-equal-bits* *v1* *v2*

**unfolding** *rewrite-preservation.simps* *rewrite-termination.simps*  
**apply** (*rule conjE*, *simp*, *simp del*: *le-expr-def*)

*BinaryFoldConstantObligation*

1. *int-and-equal-bits* *v1* *v2*  $\longrightarrow$   
 $\text{BinaryExpr } op \text{ (ConstantExpr } v1 \text{) (ConstantExpr } v2 \text{)} \sqsubseteq$   
 $\text{ConstantExpr (bin-eval } op \text{ } v1 \text{ } v2 \text{)}$
2. *int-and-equal-bits* *v1* *v2*  $\longrightarrow$   
 $\text{trm}(\text{BinaryExpr } op \text{ (ConstantExpr } v1 \text{) (ConstantExpr } v2 \text{)}) >$   
 $\text{trm}(\text{ConstantExpr (bin-eval } op \text{ } v1 \text{ } v2 \text{)})$

**using** *BinaryFoldConstant* **by** *auto*

*AddCommuteConstantRight*

**optimization** *AddCommuteConstantRight*:  $((\text{const } v) + y) \mapsto y + (\text{const } v)$  when  $\neg(\text{is-ConstantExpr } y)$

**unfolding** *rewrite-preservation.simps* *rewrite-termination.simps*  
**apply** (*rule conjE*, *simp*, *simp del*: *le-expr-def*)

*AddCommuteConstantRightObligation*

1.  $\neg \text{is-ConstantExpr } y \longrightarrow$   
 $\text{BinaryExpr BinAdd (ConstantExpr } v \text{) } y \sqsubseteq$   
 $\text{BinaryExpr BinAdd } y \text{ (ConstantExpr } v \text{)}$
2.  $\neg \text{is-ConstantExpr } y \longrightarrow$   
 $\text{trm}(\text{BinaryExpr BinAdd (ConstantExpr } v \text{) } y) >$   
 $\text{trm}(\text{BinaryExpr BinAdd } y \text{ (ConstantExpr } v \text{)})$

**using** *AddShiftConstantRight* **by** *auto*

*AddNeutral*

**optimization** *AddNeutral*:  $((e::i32\text{exp}) + (\text{const (IntVal32 0)})) \mapsto e$

**unfolding** *rewrite-preservation.simps* *rewrite-termination.simps*

**apply** (rule conjE, simp, simp del: le-expr-def)

*AddNeutralObligation*

1.  $\text{BinaryExpr BinAdd } e \ (\text{ConstantExpr } (\text{IntVal32 } 0)) \sqsubseteq e$
2.  $\text{trm}(\text{BinaryExpr BinAdd } e \ (\text{ConstantExpr } (\text{IntVal32 } 0))) > \text{trm}(e)$

**using** neutral-zero(1) rewrite-preservation.simps(1) **apply** blast  
**by** auto

*InverseLeftSub*

**optimization** *InverseLeftSub*:  $((e_1::\text{intexp}) - (e_2::\text{intexp})) + e_2 \mapsto e_1$

**unfolding** rewrite-preservation.simps rewrite-termination.simps  
**apply** (rule conjE, simp, simp del: le-expr-def)

*InverseLeftSubObligation*

1.  $\text{BinaryExpr BinAdd } (\text{BinaryExpr BinSub } e_1 \ e_2) \ e_2 \sqsubseteq e_1$
2.  $\text{trm}(\text{BinaryExpr BinAdd } (\text{BinaryExpr BinSub } e_1 \ e_2) \ e_2) > \text{trm}(e_1)$

**using** neutral-left-add-sub **by** auto

*InverseRightSub*

**optimization** *InverseRightSub*:  $(e_2::\text{intexp}) + ((e_1::\text{intexp}) - e_2) \mapsto e_1$

**unfolding** rewrite-preservation.simps rewrite-termination.simps  
**apply** (rule conjE, simp, simp del: le-expr-def)

*InverseRightSubObligation*

1.  $\text{BinaryExpr BinAdd } e_2 \ (\text{BinaryExpr BinSub } e_1 \ e_2) \sqsubseteq e_1$
2.  $\text{trm}(\text{BinaryExpr BinAdd } e_2 \ (\text{BinaryExpr BinSub } e_1 \ e_2)) > \text{trm}(e_1)$

**using** neutral-right-add-sub **by** auto

*AddToSub*

**optimization** *AddToSub*:  $-e + y \mapsto y - e$

**unfolding** rewrite-preservation.simps rewrite-termination.simps  
**apply** (rule conjE, simp, simp del: le-expr-def)

*AddToSubObligation*

1.  $\text{BinaryExpr BinAdd (UnaryExpr UnaryNeg e) y} \sqsubseteq \text{BinaryExpr BinSub y e}$
2.  $\text{trm}(\text{BinaryExpr BinAdd (UnaryExpr UnaryNeg e) y}) > \text{trm}(\text{BinaryExpr BinSub y e})$

**using** *AddLeftNegateToSub* **by** *auto*

**end**

**definition** *trm* **where** *trm* = *size*

*phase*

**phase** *AddCanonicalizations*  
**terminating** *trm*  
**begin...end**

**hide-const (open)** *Form.wf-stamp*

*phase-example*

**phase** *Conditional*  
**terminating** *trm*  
**begin**

*phase-example-1*

**optimization** *negate-condition*:  $(\neg e \text{ ? } x : y) \mapsto (e \text{ ? } y : x)$

**using** *ConditionalPhase.negate-condition*  
**by** (*auto simp: trm-def*)

*phase-example-2*

**optimization** *const-true*:  $(\text{true ? } x : y) \mapsto x$

**by** (*auto simp: trm-def*)

*phase-example-3*

**optimization** *const-false*:  $(\text{false ? } x : y) \mapsto y$

**by** (*auto simp: trm-def*)

*phase-example-4*

**optimization** *equal-branches*:  $(e \text{ ? } x : x) \mapsto x$

by (auto simp: trm-def)

*phase-example-5*

**optimization** *condition-bounds-x*:  $((u < v) ? x : y) \mapsto x$   
when (stamp-under (stamp-expr u) (stamp-expr v)  
 $\wedge$  wf-stamp u  $\wedge$  wf-stamp v)

using ConditionalPhase.condition-bounds-x(1)

by (blast, auto simp: trm-def)

*phase-example-6*

**optimization** *condition-bounds-y*:  $((x < y) ? x : y) \mapsto y$   
when (stamp-under (stamp-expr y) (stamp-expr x)  $\wedge$  wf-stamp  
x  $\wedge$  wf-stamp y)

using ConditionalPhase.condition-bounds-y(1)

by (blast, auto simp: trm-def)

*phase-example-7*

end

*termination*

$$\begin{aligned} \text{trm}(\text{UnaryExpr } op \ e) &= \text{trm}(e) + 1 \\ \text{trm}(\text{BinaryExpr } \text{BinAdd } x \ y) &= \text{trm}(x) + 2 * \text{trm}(y) \\ \text{trm}(\text{ConditionalExpr } \text{cond } t \ f) &= \text{trm}(\text{cond}) + \text{trm}(t) + \text{trm}(f) + 2 \\ \text{trm}(\text{ConstantExpr } c) &= 1 \\ \text{trm}(\text{ParameterExpr } \text{ind } s) &= 2 \\ \text{trm}(\text{LeafExpr } \text{nid } s) &= 2 \end{aligned}$$

*graph-representation*

**typedef** *IRGraph* =  $\{g :: ID \rightarrow (IRNode \times Stamp) . \text{finite } (\text{dom } g)\}$

*graph2tree*

rep:constant rep:parameter rep:conditional rep:unary rep:convert  
rep:binary rep:leaf rep:ref



## *preeval*

*is-preevaluated* (*InvokeNode* *n uu uv uw ux uy*) = *True*  
*is-preevaluated* (*InvokeWithExceptionNode* *n uz va vb vc vd ve*) = *True*  
*is-preevaluated* (*NewInstanceNode* *n vf vg vh*) = *True*  
*is-preevaluated* (*LoadFieldNode* *n vi vj vk*) = *True*  
*is-preevaluated* (*SignedDivNode* *n vl vm vn vo vp*) = *True*  
*is-preevaluated* (*SignedRemNode* *n vq vr vs vt vu*) = *True*  
*is-preevaluated* (*ValuePhiNode* *n vv vw*) = *True*  
*is-preevaluated* (*AbsNode* *v*) = *False*  
*is-preevaluated* (*AddNode* *v va*) = *False*  
*is-preevaluated* (*AndNode* *v va*) = *False*  
*is-preevaluated* (*BeginNode* *v*) = *False*  
*is-preevaluated* (*BytecodeExceptionNode* *v va vb*) = *False*  
*is-preevaluated* (*ConditionalNode* *v va vb*) = *False*  
*is-preevaluated* (*ConstantNode* *v*) = *False*  
*is-preevaluated* (*DynamicNewArrayNode* *v va vb vc vd*) = *False*  
*is-preevaluated* *EndNode* = *False*  
*is-preevaluated* (*ExceptionObjectNode* *v va*) = *False*  
*is-preevaluated* (*FrameState* *v va vb vc*) = *False*  
*is-preevaluated* (*IfNode* *v va vb*) = *False*  
*is-preevaluated* (*IntegerBelowNode* *v va*) = *False*  
*is-preevaluated* (*IntegerEqualsNode* *v va*) = *False*  
*is-preevaluated* (*IntegerLessThanNode* *v va*) = *False*  
*is-preevaluated* (*IsNullNode* *v*) = *False*  
*is-preevaluated* (*KillingBeginNode* *v*) = *False*  
*is-preevaluated* (*LeftShiftNode* *v va*) = *False*  
*is-preevaluated* (*LogicNegationNode* *v*) = *False*  
*is-preevaluated* (*LoopBeginNode* *v va vb vc*) = *False*  
*is-preevaluated* (*LoopEndNode* *v*) = *False*  
*is-preevaluated* (*LoopExitNode* *v va vb*) = *False*  
*is-preevaluated* (*MergeNode* *v va vb*) = *False*  
*is-preevaluated* (*MethodCallTargetNode* *v va*) = *False*  
*is-preevaluated* (*MulNode* *v va*) = *False*  
*is-preevaluated* (*NarrowNode* *v va vb*) = *False*  
*is-preevaluated* (*NegateNode* *v*) = *False*  
*is-preevaluated* (*NewArrayNode* *v va vb*) = *False*  
*is-preevaluated* (*NotNode* *v*) = *False*  
*is-preevaluated* (*OrNode* *v va*) = *False*  
*is-preevaluated* (*ParameterNode* *v*) = *False*  
*is-preevaluated* (*PiNode* *v va*) = *False*  
*is-preevaluated* (*ReturnNode* *v va*) = *False*  
*is-preevaluated* (*RightShiftNode* *v va*) = *False*  
*is-preevaluated* (*ShortCircuitOrNode* *v va*) = *False*  
*is-preevaluated* (*SignExtendNode* *v va vb*) = *False*

*deterministic-representation*

$$g \vdash n \simeq e_1 \wedge g \vdash n \simeq e_2 \implies e_1 = e_2$$

**thm-oracles** *repDet*

*well-formed-term-graph*

$$\exists e. g \vdash n \simeq e \wedge (\exists v. [m, p] \vdash e \mapsto v)$$

*graph-semantics*

$$([g, m, p] \vdash n \mapsto v) = (\exists e. g \vdash n \simeq e \wedge [m, p] \vdash e \mapsto v)$$

*graph-semantics-deterministic*

$$[g, m, p] \vdash n \mapsto v_1 \wedge [g, m, p] \vdash n \mapsto v_2 \implies v_1 = v_2$$

**thm-oracles** *graphDet*

**notation** (*latex*)

*graph-refinement* (*term-graph-refinement* -)

*graph-refinement*

$$\begin{aligned} \text{term-graph-refinement } g_1 \ g_2 = \\ (ids \ g_1 \subseteq ids \ g_2 \wedge \\ (\forall n. n \in ids \ g_1 \longrightarrow (\forall e. g_1 \vdash n \simeq e \longrightarrow g_2 \vdash n \sqsubseteq e))) \end{aligned}$$

**translations**

$n \leq CONST$  as-set  $n$

*graph-semantics-preservation*

$$\begin{aligned} e_1' \sqsupseteq e_2' \wedge \\ \{n\} \triangleleft g_1 \subseteq g_2 \wedge \\ g_1 \vdash n \simeq e_1' \wedge g_2 \vdash n \simeq e_2' \implies \\ \text{term-graph-refinement } g_1 \ g_2 \end{aligned}$$

**thm-oracles** *graph-semantics-preservation-subscript*

### *maximal-sharing*

$\text{maximal-sharing } g =$   
 $(\forall n_1 n_2.$   
   $n_1 \in \text{true-ids } g \wedge n_2 \in \text{true-ids } g \longrightarrow$   
   $(\forall e. g \vdash n_1 \simeq e \wedge$   
     $g \vdash n_2 \simeq e \wedge \text{stamp } g \ n_1 = \text{stamp } g \ n_2 \longrightarrow$   
     $n_1 = n_2))$

### *tree-to-graph-rewriting*

$e_1 \sqsupseteq e_2 \wedge$   
 $g_1 \vdash n \simeq e_1 \wedge$   
 $\text{maximal-sharing } g_1 \wedge$   
 $\{n\} \triangleleft g_1 \subseteq g_2 \wedge$   
 $g_2 \vdash n \simeq e_2 \wedge$   
 $\text{maximal-sharing } g_2 \implies$   
 $\text{term-graph-refinement } g_1 \ g_2$

### **thm-oracles** *tree-to-graph-rewriting*

#### *term-graph-refines-term*

$(g \vdash n \sqsubseteq e) = (\exists e'. g \vdash n \simeq e' \wedge e \sqsupseteq e')$

#### *term-graph-evaluation*

$g \vdash n \sqsubseteq e \implies \forall m \ p \ v. [m, p] \vdash e \mapsto v \longrightarrow [g, m, p] \vdash n \mapsto v$

#### *graph-construction*

$e_1 \sqsupseteq e_2 \wedge g_1 \subseteq g_2 \wedge g_2 \vdash n \simeq e_2 \implies$   
 $g_2 \vdash n \sqsubseteq e_1 \wedge \text{term-graph-refinement } g_1 \ g_2$

### **thm-oracles** *graph-construction*

#### *term-graph-reconstruction*

$g \oplus e \rightsquigarrow (g', n) \implies g' \vdash n \simeq e \wedge g \subseteq g'$

*refined-insert*

$$e_1 \sqsupseteq e_2 \wedge g_1 \oplus e_2 \rightsquigarrow (g_2, n^\wedge) \implies \\ g_2 \vdash n' \sqsubseteq e_1 \wedge \text{term-graph-refinement } g_1 \ g_2$$

**end**

**theory** *SlideSnippets*

**imports**

*Semantics.TreeToGraphThms*

*Snippets.Snipping*

**begin**

**notation** (*latex*)

*kind* ( $-\langle\!\langle\!-\!\rangle\!\rangle$ )

**notation** (*latex*)

*IRTreeEval.ord-IRExpr-inst.less-eq-IRExpr* ( $-\mapsto -$ )

*abstract-syntax-tree*

**datatype** *IRExpr* =

*UnaryExpr* *IRUnaryOp* *IRExpr*  
 | *BinaryExpr* *IRBinaryOp* *IRExpr* *IRExpr*  
 | *ConditionalExpr* *IRExpr* *IRExpr* *IRExpr*  
 | *ParameterExpr* *nat* *Stamp*  
 | *LeafExpr* *nat* *Stamp*  
 | *ConstantExpr* *Value*  
 | *ConstantVar* (*char list*)  
 | *VariableExpr* (*char list*) *Stamp*

*tree-semantic*

*semantics:constant* *semantics:parameter* *semantics:unary* *semantics:binary* *semantics:leaf*

*expression-refinement*

$$e_1 \sqsupseteq e_2 = (\forall m \ p \ v. [m,p] \vdash e_1 \mapsto v \longrightarrow [m,p] \vdash e_2 \mapsto v)$$

*graph2tree*

*semantics:constant* *semantics:unary* *semantics:binary*

#### *graph-semantics*

$$([g, m, p] \vdash n \mapsto v) = (\exists e. g \vdash n \simeq e \wedge [m, p] \vdash e \mapsto v)$$

#### *graph-refinement*

$$\begin{aligned} \text{graph-refinement } g_1 \ g_2 = \\ (\text{ids } g_1 \subseteq \text{ids } g_2 \wedge \\ (\forall n. n \in \text{ids } g_1 \longrightarrow (\forall e. g_1 \vdash n \simeq e \longrightarrow g_2 \vdash n \trianglelefteq e))) \end{aligned}$$

#### **translations**

$n \leq \text{CONST as-set } n$

#### *graph-semantics-preservation*

$$\begin{aligned} \llbracket e1' \sqsupseteq e2'; \{n'\} \triangleleft g1 \subseteq g2; \\ g1 \vdash n' \simeq e1'; g2 \vdash n' \simeq e2' \rrbracket \\ \implies \text{graph-refinement } g1 \ g2 \end{aligned}$$

#### *maximal-sharing*

$$\begin{aligned} \text{maximal-sharing } g = \\ (\forall n_1 \ n_2. \\ n_1 \in \text{true-ids } g \wedge n_2 \in \text{true-ids } g \longrightarrow \\ (\forall e. g \vdash n_1 \simeq e \wedge \\ g \vdash n_2 \simeq e \wedge \text{stamp } g \ n_1 = \text{stamp } g \ n_2 \longrightarrow \\ n_1 = n_2)) \end{aligned}$$

#### *tree-to-graph-rewriting*

$$\begin{aligned} e_1 \sqsupseteq e_2 \wedge \\ g_1 \vdash n \simeq e_1 \wedge \\ \text{maximal-sharing } g_1 \wedge \\ \{n\} \triangleleft g_1 \subseteq g_2 \wedge \\ g_2 \vdash n \simeq e_2 \wedge \text{maximal-sharing } g_2 \implies \\ \text{graph-refinement } g_1 \ g_2 \end{aligned}$$

*graph-represents-expression*

$$(g \vdash n \trianglelefteq e) = (\exists e'. g \vdash n \simeq e' \wedge e \sqsupseteq e')$$

*graph-construction*

$$e_1 \sqsupseteq e_2 \wedge g_1 \subseteq g_2 \wedge g_2 \vdash n \simeq e_2 \implies \\ g_2 \vdash n \trianglelefteq e_1 \wedge \text{graph-refinement } g_1 \ g_2$$

**end**