Veriopt Theories

September 11, 2022

Contents

1 Verifying term graph optimizations using Isabelle/HOL

theory *TreeSnippets*
  imports
    Canonicalizations.ConditionalPhase
    Optimizations.CanonicalizationSyntax
    Semantics.TreeToGraphThms
    Snippets.Snipping
    HOL−Library.OptionalSugar
begin

— First, we disable undesirable markup.
declare [[*show-types=false*,*show-sorts=false*]]
no-notation *ConditionalExpr* (- *?* - : -)
translations
  n <= *CONST Rep-intexp n*
  n <= *CONST Rep-i32exp n*

1.1 Markup syntax for common operations

notation (*latex*)
  kind (-⟪-⟫)

notation (*latex*)
  valid-value (- ∈ -)

notation (*latex*)
  val-to-bool (*bool-of* -)

notation (*latex*)
constantAsStamp (*stamp-from-value* -)

notation (*latex*)
size (*trm(-)*)

1.2 Representing canonicalization optimizations

We wish to provide an example of the semantics layers at which optimizations can be expressed.

thm *diff-self*
thm *diff-diff-cancel*

algebraic-laws

$x - x = (0 :: 'a1)$
$y \sqsubseteq x \Longrightarrow x - (x - y) = y$

lemma *diff-self-value*: $\forall v{::}int64.\ v - v = 0$
  by *simp*
lemma *diff-diff-cancel-value*:
  $\forall\ (v_1{::}int64)\ (v_2{::}int64)\ .\ v_1 - (v_1 - v_2) = v_2$
  by *simp*

algebraic-laws-values

$\forall v :: 64\ word.\ v - v = (0 :: 64\ word)$
$\forall (v_1{::}64\ word)\ v_2 :: 64\ word.\ v_1 - (v_1 - v_2) = v_2$

translations
  n <= *CONST ConstantExpr* (*CONST IntVal b n*)
  x − *y* <= *CONST BinaryExpr* (*CONST BinSub*) *x y*
notation (*ExprRule* **output**)
  Refines (- ⟼ -)
lemma *diff-self-expr*:
  assumes $\forall\ m\ p\ v.\ [m,p] \vdash exp[e - e] \mapsto IntVal\ b\ v$
  shows $exp[e - e] \geq exp[const\ (IntVal\ b\ 0)]$
  using *assms* **apply** *simp*
  by (*metis*(*full-types*) *evalDet val-to-bool.simps*(*1*) *zero-neq-one*)

lemma *diff-diff-cancel-expr*:
  shows $exp[e_1 - (e_1 - e_2)] \geq exp[e_2]$
  apply *simp* **sorry**

algebraic-laws-expressions

$e - e \longmapsto 0$
$e_1 - (e_1 - e_2) \longmapsto e_2$

no-translations
n <= *CONST ConstantExpr* (*CONST IntVal b n*)
$x - y$ <= *CONST BinaryExpr* (*CONST BinSub*) *x y*

definition *wf-stamp* :: *IRExpr* ⇒ *bool* **where**
wf-stamp e = (∀ *m p v*. ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *valid-value v* (*stamp-expr e*))

lemma *wf-stamp-eval*:
assumes *wf-stamp e*
assumes *stamp-expr e* = *IntegerStamp b lo hi*
shows ∀ *m p v*. ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ (∃ *vv*. *v* = *IntVal b vv*)
using *assms* **unfolding** *wf-stamp-def*
using *valid-int-same-bits valid-int*
by *metis*

phase *SnipPhase*
terminating *size*
begin
lemma *sub-same-val*:
assumes *val*[*e* − *e*] = *IntVal b v*
shows *val*[*e* − *e*] = *val*[*IntVal b 0*]
using *assms* **by** (*cases e*; *auto*)

sub-same-32

optimization *sub-same-32*:
(*e* − *e*) ⟼ *const* (*IntVal b 0*)
when ((*stamp-expr exp*[*e* − *e*] = *IntegerStamp b lo hi*) ∧ *wf-stamp exp*[*e* − *e*])

apply *simp*
apply (*metis Suc-lessI add-is-1 add-pos-pos size-gt-0*)
apply (*rule impI*) **apply** *simp*
proof −
assume *assms*: *stamp-binary BinSub* (*stamp-expr e*) (*stamp-expr e*) = *IntegerStamp b lo hi* ∧ *wf-stamp exp*[*e* − *e*]
have ∀ *m p v* . ([*m*, *p*] ⊢ *exp*[*e* − *e*] ↦ *v*) ⟶ (∃ *vv*. *v* = *IntVal b vv*)
using *assms wf-stamp-eval*
by (*metis stamp-expr.simps*(*2*))
then show ∀ *m p v*. ([*m*,*p*] ⊢ *BinaryExpr BinSub e e* ↦ *v*) ⟶ ([*m*,*p*] ⊢ *ConstantExpr* (*IntVal b 0*) ↦ *v*)
by (*smt* (*verit*, *best*) *BinaryExprE TreeSnippets.wf-stamp-def assms bin-eval.simps*(*3*) *constantAsStamp.simps*(*1*) *evalDet stamp-expr.simps*(*2*) *sub-same-val unfold-const*

valid-stamp.simps(1) valid-value.simps(1))
qed
thm-oracles *sub-same-32*
end

1.3 Representing terms

We wish to show a simple example of expressions represented as terms.

ast-example

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

Then we need to show the datatypes that compose the example expression.

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
  | IntVal nat (64 word)
  | ObjRef (nat option)
  | ObjStr (char list)
```

1.4 Term semantics

The core expression evaluation functions need to be introduced.

eval

unary-eval :: *IRUnaryOp* ⇒ *Value* ⇒ *Value*

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

We then provide the full semantics of IR expressions.

no-translations
$(prop)\ P \wedge Q \Longrightarrow R <= (prop)\ P \Longrightarrow Q \Longrightarrow R$
translations
$(prop)\ P \Longrightarrow Q \Longrightarrow R <= (prop)\ P \wedge Q \Longrightarrow R$

tree-semantics

semantics:unary semantics:binary semantics:conditional semantics:constant semantics:parameter semantics:leaf

no-translations
$(prop)\ P \Longrightarrow Q \Longrightarrow R <= (prop)\ P \wedge Q \Longrightarrow R$
translations
$(prop)\ P \wedge Q \Longrightarrow R <= (prop)\ P \Longrightarrow Q \Longrightarrow R$

And show that expression evaluation is deterministic.

tree-evaluation-deterministic

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

We then want to start demonstrating the obligations for optimizations. For this we define refinement over terms.

expression-refinement

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

To motivate this definition we show the obligations generated by optimization definitions.

phase *SnipPhase*
terminating *size*
begin

InverseLeftSub

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

InverseLeftSubObligation

1. *trm(*e_1*)* $<$ *trm(BinaryExpr BinAdd* (*BinaryExpr BinSub* e_1 e_2) e_2*)*
2. *BinaryExpr BinAdd* (*BinaryExpr BinSub* e_1 e_2) $e_2 \sqsupseteq e_1$

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

InverseRightSub

optimization *InverseRightSub*: $(e_2::intexp) + ((e_1::intexp) - e_2) \longmapsto e_1$

InverseRightSubObligation

1. *trm(*e_1*)* $<$ *trm(BinaryExpr BinAdd* e_2 (*BinaryExpr BinSub* e_1 e_2)*)*
2. *BinaryExpr BinAdd* e_2 (*BinaryExpr BinSub* e_1 e_2) $\sqsupseteq e_1$

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

expression-refinement-monotone

$$\begin{array}{ll} e \sqsupseteq e' & \Longrightarrow \textit{UnaryExpr op } e \sqsupseteq \textit{UnaryExpr op } e' \\ x \sqsupseteq x' \wedge y \sqsupseteq y' & \Longrightarrow \textit{BinaryExpr op } x\ y \sqsupseteq \textit{BinaryExpr op } x'\ y' \\ ce \sqsupseteq ce' \wedge te \sqsupseteq te' \wedge fe \sqsupseteq fe' & \Longrightarrow \textit{ConditionalExpr } ce\ te\ fe \sqsupseteq \textit{ConditionalExpr } ce'\ te'\ fe' \end{array}$$

phase *SnipPhase*
terminating *size*
begin

BinaryFoldConstant

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

BinaryFoldConstantObligation

1. *int-and-equal-bits v1 v2* $\longrightarrow$
   trm(ConstantExpr (*bin-eval op v1 v2*)*)*
   $<$ *trm(BinaryExpr op* (*ConstantExpr v1*) (*ConstantExpr v2*)*)*
2. *int-and-equal-bits v1 v2* $\longrightarrow$
   BinaryExpr op (*ConstantExpr v1*) (*ConstantExpr v2*) $\sqsupseteq$
   ConstantExpr (*bin-eval op v1 v2*)

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

AddCommuteConstantRight

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

AddCommuteConstantRightObligation

1. $\neg$ *is-ConstantExpr y* $\longrightarrow$
   trm(BinaryExpr BinAdd y (*ConstantExpr v*)*)*
   $<$ *trm(BinaryExpr BinAdd* (*ConstantExpr v*) *y)*
2. $\neg$ *is-ConstantExpr y* $\longrightarrow$
   BinaryExpr BinAdd (*ConstantExpr v*) *y* $\sqsupseteq$
   BinaryExpr BinAdd y (*ConstantExpr v*)

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

AddNeutral

optimization *AddNeutral*: $((e{::}i32exp) + (const\ (IntVal\ 32\ 0))) \longmapsto e$

AddNeutralObligation

1. *trm(e)* $<$ *trm(BinaryExpr BinAdd e* (*ConstantExpr* (*IntVal 32 0*))*)*
2. *BinaryExpr BinAdd e* (*ConstantExpr* (*IntVal 32 0*)) $\sqsupseteq$ *e*

apply (*rule conjE*, *simp*, *simp del*: *le-expr-def*)
using *neutral-zero*(*1*) *rewrite-preservation.simps*(*1*) **by** *blast*

AddToSub

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

AddToSubObligation

1. *trm(BinaryExpr BinSub y e)* $<$ *trm(BinaryExpr BinAdd* (*UnaryExpr UnaryNeg e*) *y)*
2. *BinaryExpr BinAdd* (*UnaryExpr UnaryNeg e*) *y* $\sqsupseteq$ *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*: $((!e)\ ?\ x : y) \longmapsto (e\ ?\ y : x)$

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

phase-example-2

optimization *const-true*: $(true\ ?\ x : y) \longmapsto x$

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

phase-example-3

optimization *const-false*: $(false\ ?\ x : y) \longmapsto y$

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

phase-example-4

optimization *equal-branches*: $(e\ ?\ x : x) \longmapsto x$

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

phase-example-7

end

termination

$$\begin{array}{lcl}
\mathit{trm(UnaryExpr\ op\ e)} & = & \mathit{trm(e) + 1} \\
\mathit{trm(BinaryExpr\ BinAdd\ x\ y)} & = & \mathit{trm(x) + 2 * trm(y)} \\
\mathit{trm(BinaryExpr\ BinXor\ x\ y)} & = & \mathit{trm(x) + trm(y)} \\
\mathit{trm(ConditionalExpr\ cond\ t\ f)} & = & \mathit{trm(cond) + trm(t) + trm(f) + 2} \\
\mathit{trm(ConstantExpr\ c)} & = & \mathit{1} \\
\mathit{trm(ParameterExpr\ ind\ s)} & = & \mathit{2}
\end{array}$$

graph-representation

typedef IRGraph =
$\{g :: \mathit{ID} \rightharpoonup (\mathit{IRNode} \times \mathit{Stamp}) \;.\; \mathit{finite}\ (\mathit{dom}\ g)\}$

no-translations
$(\mathit{prop})\ P \wedge Q \Longrightarrow R <= (\mathit{prop})\ P \Longrightarrow Q \Longrightarrow R$
translations
$(\mathit{prop})\ P \Longrightarrow Q \Longrightarrow R <= (\mathit{prop})\ P \wedge Q \Longrightarrow R$

graph2tree

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

no-translations
$(\mathit{prop})\ P \Longrightarrow Q \Longrightarrow R <= (\mathit{prop})\ P \wedge Q \Longrightarrow R$
translations
$(\mathit{prop})\ P \wedge Q \Longrightarrow R <= (\mathit{prop})\ P \Longrightarrow Q \Longrightarrow R$

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

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

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

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

thm-oracles *graphDet*

notation (*latex*)
graph-refinement (*term-graph-refinement* -)

graph-refinement

term-graph-refinement $g_1\ g_2 =$
$(\mathit{ids}\ g_1 \subseteq \mathit{ids}\ g_2 \wedge$
$(\forall\, n.\ n \in \mathit{ids}\ g_1 \longrightarrow (\forall\, e.\ g_1 \vdash n \simeq e \longrightarrow g_2 \vdash n \trianglelefteq e)))$

translations
n <= *CONST as-set* n

graph-semantics-preservation

$e_1{}' \sqsupseteq e_2{}' \wedge$
$\{n\} \mathrel{\lhd\!\!\!\!-} g_1 \subseteq g_2 \wedge$
$g_1 \vdash n \simeq e_1{}' \wedge g_2 \vdash n \simeq e_2{}' \Longrightarrow$
term-graph-refinement $g_1\ g_2$

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

$$
\begin{array}{l}
\textit{maximal-sharing}\ g = \\
(\forall\, n_1\ n_2. \\
\quad n_1 \in \textit{true-ids}\ g \wedge n_2 \in \textit{true-ids}\ g \longrightarrow \\
\quad (\forall\, e.\ g \vdash n_1 \simeq e \wedge \\
\qquad\quad g \vdash n_2 \simeq e \wedge \textit{stamp}\ g\ n_1 = \textit{stamp}\ g\ n_2 \longrightarrow \\
\qquad\quad n_1 = n_2))
\end{array}
$$

tree-to-graph-rewriting

$$
\begin{array}{l}
e_1 \sqsupseteq e_2 \wedge \\
g_1 \vdash n \simeq e_1 \wedge \\
\textit{maximal-sharing}\ g_1 \wedge \\
\{n\} \trianglelefteq g_1 \subseteq g_2 \wedge \\
g_2 \vdash n \simeq e_2 \wedge \\
\textit{maximal-sharing}\ g_2 \Longrightarrow \\
\textit{term-graph-refinement}\ g_1\ g_2
\end{array}
$$

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

term-graph-refines-term

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

term-graph-evaluation

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

graph-construction

$$
\begin{array}{l}
e_1 \sqsupseteq e_2 \wedge g_1 \subseteq g_2 \wedge g_2 \vdash n \simeq e_2 \Longrightarrow \\
g_2 \vdash n \trianglelefteq e_1 \wedge \textit{term-graph-refinement}\ g_1\ g_2
\end{array}
$$

thm-oracles *graph-construction*

term-graph-reconstruction

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

refined-insert

$e_1 \sqsupseteq e_2 \wedge g_1 \oplus e_2 \rightsquigarrow (g_2,\ n') \Longrightarrow$
$g_2 \vdash n' \trianglelefteq e_1 \wedge \textit{term-graph-refinement}\ g_1\ g_2$

end
theory *SlideSnippets*
imports
Semantics.TreeToGraphThms
Snippets.Snipping
begin

notation (*latex*)
kind (-⟪-⟫)

notation (*latex*)
IRTreeEval.ord-IRExpr-inst.less-eq-IRExpr (- $\longmapsto$ -)

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-semantics

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

expression-refinement

$$(e_1{::}\textit{IRExpr}) \sqsupseteq (e_2{::}\textit{IRExpr}) = (\forall (m{::}\textit{nat} \Rightarrow \textit{Value})\ (p{::}\textit{Value list})\ v{::}\textit{Value}.\ [m,p] \vdash e_1 \mapsto v \longrightarrow [m,p] \vdash e_2 \mapsto v)$$

graph2tree

semantics:constant semantics:unary semantics:binary *graph-semantics*

$$([g{::}IRGraph, m{::}nat \Rightarrow Value, p{::}Value\ list] \vdash n{::}nat \mapsto v{::}Value) = (\exists\, e{::}IRExpr.\ g \vdash n \simeq e \wedge [m,p] \vdash e \mapsto v)$$

graph-refinement

$graph\text{-}refinement\ (g_1{::}IRGraph)\ (g_2{::}IRGraph) =$
$(ids\ g_1 \subseteq ids\ g_2 \wedge$
$(\forall\, n{::}nat.$
$n \in ids\ g_1 \longrightarrow (\forall\, e{::}IRExpr.\ g_1 \vdash n \simeq e \longrightarrow g_2 \vdash n \trianglelefteq e)))$

translations

$n <= CONST\ as\text{-}set\ n$

graph-semantics-preservation

$[\![(e1'{::}IRExpr) \sqsupseteq$
$(e2'{::}IRExpr);$
$\{n'{::}nat\} \trianglelefteq g1{::}IRGraph$
$\subseteq (g2{::}IRGraph);$
$g1 \vdash n' \simeq e1';\ g2 \vdash n' \simeq e2']\!]$
$\Longrightarrow graph\text{-}refinement\ g1\ g2$

maximal-sharing

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

tree-to-graph-rewriting

$(e_1::IRExpr) \sqsupseteq (e_2::IRExpr) \wedge$
$g_1::IRGraph \vdash n::nat \simeq e_1 \wedge$
$maximal\text{-}sharing\ g_1 \wedge$
$\{n\} \trianglelefteq g_1 \subseteq (g_2::IRGraph) \wedge$
$g_2 \vdash n \simeq e_2 \wedge maximal\text{-}sharing\ g_2 \Longrightarrow$
$graph\text{-}refinement\ g_1\ g_2$

graph-represents-expression

$(g::IRGraph \vdash n::nat \trianglelefteq e::IRExpr) = (\exists\, e'::IRExpr.\ g \vdash n \simeq e' \wedge e \sqsupseteq e')$

graph-construction

$(e_1::IRExpr) \sqsupseteq (e_2::IRExpr) \wedge$
$(g_1::IRGraph) \subseteq (g_2::IRGraph) \wedge$
$g_2 \vdash n::nat \simeq e_2 \Longrightarrow$
$g_2 \vdash n \trianglelefteq e_1 \wedge graph\text{-}refinement\ g_1\ g_2$

end