Veriopt Theories

September 14, 2022

Contents

1 Verifying term graph optimizations using Isabelle/HOL

```
theory TreeSnippets
  imports
    Canonicalizations.BinaryNode
    Canonicalizations.ConditionalPhase
    Canonicalizations.AddPhase
    Semantics.TreeToGraphThms
    Snippets.Snipping
    HOL−Library.OptionalSugar
begin

— First, we disable undesirable markup.
declare [[show-types=false,show-sorts=false]]
no-notation ConditionalExpr (- ? - : -)
```

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.

lemma *diff-self*:
fixes x :: *int*
shows $x - x = 0$
by *simp*
lemma *diff-diff-cancel*:
fixes $x\ y$ :: *int*
shows $x - (x - y) = y$
by *simp*
thm *diff-self*
thm *diff-diff-cancel*

algebraic-laws

$$x - x = 0 \tag{1}$$

$$x - (x - y) = y \tag{2}$$

lemma *diff-self-value*: $\forall\, v{::}'a{::}len\ word.\ v - v = 0$
by *simp*
lemma *diff-diff-cancel-value*:
$\forall\ v_1\ v_2{::}'a{::}len\ word\ .\ v_1 - (v_1 - v_2) = v_2$
by *simp*

algebraic-laws-values

$$\forall\, v :: {}'a\ word.\ v - v = (0 :: {}'a\ word) \tag{3}$$

$$\forall\, (v_1{::}'a\ word)\ v_2 :: {}'a\ word.\ v_1 - (v_1 - v_2) = v_2 \tag{4}$$

translations
n <= *CONST ConstantExpr* (*CONST IntVal* b n)
$x - y$ <= *CONST BinaryExpr* (*CONST BinSub*) x y
notation (*ExprRule* **output**)
Refines (- $\longmapsto$ -)
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)

method open-eval = (simp; (rule impI)?; (rule allI)+; rule impI)

lemma diff-diff-cancel-expr:
  shows exp[e_1 − (e_1 − e_2)] ≥ exp[e_2]
  apply open-eval
  subgoal premises eval for m p v
  proof −
    obtain v1 where v1: [m, p] ⊢ e_1 ↦ v1
      using eval by blast
    obtain v2 where v2: [m, p] ⊢ e_2 ↦ v2
      using eval by blast
    then have e: [m, p] ⊢ exp[e_1 − (e_1 − e_2)] ↦ val[v1 − (v1 − v2)]
      using v1 v2 eval
      by (smt (verit, ccfv-SIG) bin-eval.simps(3) evalDet unfold-binary)
    then have notUn: val[v1 − (v1 − v2)] ≠ UndefVal
      using evaltree-not-undef by auto
    then have val[v1 − (v1 − v2)] = v2
      apply (cases v1; cases v2; auto simp: notUn)
      using eval-unused-bits-zero v2 apply blast
      by (metis(full-types) intval-sub.simps(5))
    then show ?thesis
      by (metis e eval evalDet v2)
  qed
  done

thm-oracles diff-diff-cancel-expr
```

algebraic-laws-expressions

$$e - e \longmapsto 0 \tag{5}$$

$$e_1 - (e_1 - e_2) \longmapsto e_2 \tag{6}$$

```
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 SubIdentity:
  (e − e) ⟼ ConstantExpr (IntVal b 0)
     when ((stamp-expr exp[e − e] = IntegerStamp b lo hi) ∧ wf-stamp exp[e
− e])
```

```
 apply (metis Suc-lessI mult-eq-1-iff mult-pos-pos nat.discI nat.inject numeral-3-eq-3
size-pos zero-less-numeral)
  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 SubIdentity
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*:
  $(e_1 - e_2) + e_2 \longmapsto e_1$

InverseLeftSubObligation

1. $BinaryExpr\ BinAdd\ (BinaryExpr\ BinSub\ e_1\ e_2)\ e_2 \sqsupseteq e_1$

  using *RedundantSubAdd* **by** *auto*

InverseRightSub

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

InverseRightSubObligation

1. $BinaryExpr\ BinAdd\ e_2\ (BinaryExpr\ BinSub\ e_1\ e_2) \sqsupseteq e_1$

  using *RedundantSubAdd2(1) rewrite-preservation.simps(1)* **by** *blast*
end

expression-refinement-monotone

$e \sqsupseteq e' \Longrightarrow \mathit{UnaryExpr}\ \mathit{op}\ e \sqsupseteq \mathit{UnaryExpr}\ \mathit{op}\ e'$

$x \sqsupseteq x' \wedge y \sqsupseteq y' \Longrightarrow \mathit{BinaryExpr}\ \mathit{op}\ x\ y \sqsupseteq \mathit{BinaryExpr}\ \mathit{op}\ x'\ y'$

$\mathit{ce} \sqsupseteq \mathit{ce}' \wedge \mathit{te} \sqsupseteq \mathit{te}' \wedge \mathit{fe} \sqsupseteq \mathit{fe}' \Longrightarrow$
$\mathit{ConditionalExpr}\ \mathit{ce}\ \mathit{te}\ \mathit{fe} \sqsupseteq \mathit{ConditionalExpr}\ \mathit{ce}'\ \mathit{te}'\ \mathit{fe}'$

phase *SnipPhase*
terminating *size*
begin

BinaryFoldConstant

optimization *BinaryFoldConstant*: $\mathit{BinaryExpr}\ \mathit{op}\ (\mathit{const}\ \mathit{v1})\ (\mathit{const}\ \mathit{v2})$
$\longmapsto \mathit{ConstantExpr}\ (\mathit{bin\text{-}eval}\ \mathit{op}\ \mathit{v1}\ \mathit{v2})$

BinaryFoldConstantObligation

1. $\mathit{BinaryExpr}\ \mathit{op}\ (\mathit{ConstantExpr}\ \mathit{v1})\ (\mathit{ConstantExpr}\ \mathit{v2}) \sqsupseteq$
   $\mathit{ConstantExpr}\ (\mathit{bin\text{-}eval}\ \mathit{op}\ \mathit{v1}\ \mathit{v2})$

using *BinaryFoldConstant(1)* **by** *auto*

AddCommuteConstantRight

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

AddCommuteConstantRightObligation

1. $\neg\ \mathit{is\text{-}ConstantExpr}\ y \longrightarrow \mathit{Suc}\ 0 < \mathit{trm(y)}$
2. $\neg\ \mathit{is\text{-}ConstantExpr}\ y \longrightarrow$
   $\mathit{BinaryExpr}\ \mathit{BinAdd}\ (\mathit{ConstantExpr}\ v)\ y \sqsupseteq$
   $\mathit{BinaryExpr}\ \mathit{BinAdd}\ y\ (\mathit{ConstantExpr}\ v)$

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

AddNeutral

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

AddNeutralObligation

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

using *AddNeutral*(*1*) *rewrite-preservation.simps*(*1*) **by** *blast*

AddToSub

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

AddToSubObligation

1. BinaryExpr BinAdd (*UnaryExpr UnaryNeg e*) *y* $\sqsupseteq$ *BinaryExpr BinSub y e*

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

end

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

lemma *trm-defn*[*size-simps*]:
trm x = *size x*
by (*simp add*: *trm-def*)

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*) *when* (*wf-stamp e* $\wedge$ *stamp-expr e* = *IntegerStamp b lo hi* $\wedge$ *b* > *0*)

apply (*simp add*: *size-simps*)
using *ConditionalPhase.NegateConditionFlipBranches*(*1*)
using *StampEvalThms.wf-stamp-def TreeSnippets.wf-stamp-def* **by** *force*

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-5

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

apply (*auto simp*: *trm-def*)
using *ConditionalPhase.condition-bounds-x*(*1*)
by (*metis*(*full-types*) *StampEvalThms.wf-stamp-def TreeSnippets.wf-stamp-def bin-eval.simps*(*12*) *stamp-under-defn*)

phase-example-6

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

apply (*auto simp*: *trm-def*)
using *ConditionalPhase.condition-bounds-y*(*1*)
by (*metis*(*full-types*) *StampEvalThms.wf-stamp-def TreeSnippets.wf-stamp-def bin-eval.simps*(*12*) *stamp-under-defn-inverse*)

phase-example-7

end

thm *size.simps*

termination

$\mathit{trm}(\mathit{UnaryExpr}\ \mathit{op}\ e) = \mathit{trm}(e) * 2$

$\mathit{trm}(\mathit{BinaryExpr}\ \mathit{op}\ x\ y) = \mathit{trm}(x) + \mathit{trm}(y) * 2$

$\mathit{trm}(\mathit{ConditionalExpr}\ \mathit{cond}\ t\ f) = \mathit{trm}(\mathit{cond}) + \mathit{trm}(t) + \mathit{trm}(f) + 2$

$\mathit{trm}(\mathit{ConstantExpr}\ c) = 1$

$\mathit{trm}(\mathit{ParameterExpr}\ \mathit{ind}\ s) = 2$

$\mathit{trm}(\mathit{LeafExpr}\ \mathit{nid}\ s) = 2$

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 $g_1\ g_2 =$
$(ids\ g_1 \subseteq ids\ g_2 \wedge$
$(\forall\, n.\ n \in ids\ g_1 \longrightarrow (\forall\, e.\ g_1 \vdash n \simeq e \longrightarrow g_2 \vdash n \trianglelefteq e)))$

translations
n <= *CONST as-set* n

graph-semantics-preservation

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

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

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

tree-to-graph-rewriting

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

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

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

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

$e_1 \sqsupseteq e_2 \wedge g_1 \oplus e_2 \leadsto (g_2,\ n') \Longrightarrow$
$g_2 \vdash n' \trianglelefteq e_1 \wedge$ *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 (- ⟼ -)

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

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

graph2tree

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

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

graph-refinement

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

translations

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

graph-semantics-preservation

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

maximal-sharing

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

tree-to-graph-rewriting

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

graph-represents-expression

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

graph-construction

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

end