Veriopt Theories

September 6, 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
  $y > x <= x < y$

notation (*latex*)
  greater (- >/ -)

translations

$n <= CONST\ Rep\text{-}intexp\ n$
$n <= CONST\ Rep\text{-}i32exp\ n$

lemma *vminusv*: $\forall\ vv\ v\ .\ vv = IntVal\ 32\ v \longrightarrow v - v = 0$
by *simp*
thm-oracles *vminusv*

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

lemma *redundant-sub*:
$\forall\ vv_1\ vv_2\ v_1\ v_2\ .\ vv_1 = IntVal\ 32\ v_1 \wedge vv_2 = IntVal\ 32\ v_2 \longrightarrow v_1 - (v_1 - v_2) = v_2$
by *simp*
thm-oracles *redundant-sub*

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

val-eq

$\forall\ (vv{::}Value)\ v :: 64\ word.\ vv = IntVal\ (32 :: nat)\ v \longrightarrow v - v = (0 :: 64\ word)$

$\forall\ (vv_1{::}Value)\ (vv_2{::}Value)\ (v_1{::}64\ word)\ v_2 :: 64\ word.\ vv_1 = IntVal\ (32 :: nat)\ v_1 \wedge vv_2 = IntVal\ (32 :: nat)\ v_2 \longrightarrow v_1 - (v_1 - v_2) = v_2$

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*)

definition *wf-stamp* :: *IRExpr* ⇒ *bool* **where**
wf-stamp e = $(\forall\ m\ p\ v.\ ([m, p] \vdash e \mapsto v) \longrightarrow valid\text{-}value\ v\ (stamp\text{-}expr\ e))$

lemma *wf-stamp-eval*:
assumes *wf-stamp e*
assumes *stamp-expr e* = *IntegerStamp b lo hi*
shows $\forall\ m\ p\ v.\ ([m, p] \vdash e \mapsto v) \longrightarrow (\exists\ vv.\ v = IntVal\ b\ vv)$
using *assms* **unfolding** *wf-stamp-def*
using *valid-int-same-bits valid-int*
by *metis*

phase *tmp*
terminating *size*
begin

sub-same-32

optimization *sub-same-32*: $((e::i32exp) - e) \longmapsto const\ (IntVal\ b\ 0)$
when $((stamp\text{-}expr\ exp[e - e] = IntegerStamp\ b\ lo\ hi) \wedge wf\text{-}stamp\ exp[e - e])$

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

ast-example

BinaryExpr BinAdd (*BinaryExpr BinMul* (*x* :: *IRExpr*) *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*
| *IntVal nat* (*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 semantics:constant semantics:parameter semantics:leaf

tree-evaluation-deterministic

[*m* :: *nat* ⇒ *Value*,*p* :: *Value list*] ⊢ *e* :: *IRExpr* ↦ v_1 :: *Value* ∧
[*m*,*p*] ⊢ *e* ↦ v_2 :: *Value* ⟹
$v_1 = v_2$

thm-oracles *evalDet*

expression-refinement

(e_1 :: *IRExpr*) ⊒ (e_2 :: *IRExpr*) = (∀ (*m*::*nat* ⇒ *Value*) (*p*:: *Value list*)
v :: *Value*. [*m*,*p*] ⊢ e_1 ↦ *v* ⟶ [*m*,*p*] ⊢ e_2 ↦ *v*)

expression-refinement-monotone

(e :: *IRExpr*) ⊒ (e' :: *IRExpr*)
(x :: *IRExpr*) ⊒ (x' :: *IRExpr*) ∧ (y :: *IRExpr*) ⊒ (y' :: *IRExpr*)
(ce :: *IRExpr*) ⊒ (ce' :: *IRExpr*) ∧ (te :: *IRExpr*) ⊒ (te' :: *IRExpr*) ∧ (fe :: *IRExpr*) ⊒ (fe' :: *I*

ML ‹
(∗*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*: *BinaryExpr op* (*const v1*) (*const v2*)
⟼ *ConstantExpr* (*bin-eval op v1 v2*) *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* $\longrightarrow$
   trm(BinaryExpr op (ConstantExpr v1) (ConstantExpr v2)) > Suc (0 :: nat)
2. *int-and-equal-bits v1 v2* $\longrightarrow$
   BinaryExpr op (ConstantExpr v1) (ConstantExpr v2) $\sqsupseteq$
   ConstantExpr (bin-eval op v1 v2)

variables:
op :: *IRBinaryOp*
v1, *v2* :: *Value*

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

AddCommuteConstantRight

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

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

AddCommuteConstantRightObligation

1. $\neg$ *is-ConstantExpr y* $\longrightarrow$ *trm(y) > Suc (0 :: nat)*
2. $\neg$ *is-ConstantExpr y* $\longrightarrow$
   BinaryExpr BinAdd (ConstantExpr v) y $\sqsupseteq$
   BinaryExpr BinAdd y (ConstantExpr v)

variables:
v :: *Value*
y :: *IRExpr*

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

AddNeutral

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

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

AddNeutralObligation

1. BinaryExpr BinAdd e (ConstantExpr (IntVal (32 :: nat) (0 :: 64 word)))
$\sqsupseteq$
e
variables:
e :: i32exp

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

InverseLeftSub

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

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

InverseLeftSubObligation

1. trm(e₂) > 0 :: nat
2. BinaryExpr BinAdd (BinaryExpr BinSub e₁ e₂) e₂ ⊒ e₁
variables:
e₁, e₂ :: intexp

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

InverseRightSub

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

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

InverseRightSubObligation

1. trm(e₁) > 0 :: nat ∨ trm(e₂) > 0 :: nat
2. BinaryExpr BinAdd e₂ (BinaryExpr BinSub e₁ e₂) ⊒ e₁
variables:
e₁, e₂ :: intexp

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

AddToSub

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

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

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

AddToSubObligation

1. BinaryExpr BinAdd (*UnaryExpr UnaryNeg e*) *y* ⊒ *BinaryExpr BinSub y e*
variables:
e, *y* :: *IRExpr*

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*) ⟼ (*e ? y* : *x*)

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

phase-example-2

optimization *const-true*: (*true ? x* : *y*) ⟼ *x*

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

phase-example-3

optimization *const-false*: (*false ? x* : *y*) ⟼ *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

$$
\begin{aligned}
&trm(UnaryExpr\ (op :: IRUnaryOp)\ (e :: IRExpr)) && = trm(e :: IRExpr) + (1\\
&trm(BinaryExpr\ BinAdd\ (x :: IRExpr)\ (y :: IRExpr)) && = trm(x :: IRExpr) + (2\\
&trm(BinaryExpr\ BinIntegerBelow\ (x :: IRExpr)\ (y :: IRExpr)) && = trm(x :: IRExpr) + trm\\
&trm(ConditionalExpr\ (cond :: IRExpr)\ (t :: IRExpr)\ (f :: IRExpr)) && = trm(cond :: IRExpr) +\\
&trm(ConstantExpr\ (c :: Value)) && = 1 :: nat\\
&trm(ParameterExpr\ (ind :: nat)\ (s :: Stamp)) && = 2 :: nat
\end{aligned}
$$

graph-representation

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

graph2tree

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

preeval

is-preevaluated (*InvokeNode* (*n* :: *nat*) (*uu* :: *nat*) (*uv* :: *nat option*) (*uw* :: *nat option*) (*ux* :: *nat option*) (*uy* :: *nat*)) = *True*

is-preevaluated (*InvokeWithExceptionNode* (*n* :: *nat*) (*uz* :: *nat*) (*va* :: *nat option*) (*vb* :: *nat option*) (*vc* :: *nat option*) (*vd* :: *nat*) (*ve* :: *nat*)) = *True*

is-preevaluated (*NewInstanceNode* (*n* :: *nat*) (*vf* :: *char list*) (*vg* :: *nat option*) (*vh* :: *nat*)) = *True*

is-preevaluated (*LoadFieldNode* (*n* :: *nat*) (*vi* :: *char list*) (*vj* :: *nat option*) (*vk* :: *nat*)) = *True*

is-preevaluated (*SignedDivNode* (*n* :: *nat*) (*vl* :: *nat*) (*vm* :: *nat*) (*vn* :: *nat option*) (*vo* :: *nat option*) (*vp* :: *nat*)) = *True*

is-preevaluated (*SignedRemNode* (*n* :: *nat*) (*vq* :: *nat*) (*vr* :: *nat*) (*vs* :: *nat option*) (*vt* :: *nat option*) (*vu* :: *nat*)) = *True*

is-preevaluated (*ValuePhiNode* (*n* :: *nat*) (*vv* :: *nat list*) (*vw* :: *nat*)) = *True*

is-preevaluated (*AbsNode* (*v* :: *nat*)) = *False*

is-preevaluated (*AddNode* (*v* :: *nat*) (*va* :: *nat*)) = *False*

is-preevaluated (*AndNode* (*v* :: *nat*) (*va* :: *nat*)) = *False*

is-preevaluated (*BeginNode* (*v* :: *nat*)) = *False*

is-preevaluated (*BytecodeExceptionNode* (*v* :: *nat list*) (*va* :: *nat option*) (*vb* :: *nat*)) = *False*

is-preevaluated (*ConditionalNode* (*v* :: *nat*) (*va* :: *nat*) (*vb* :: *nat*)) = *False*

is-preevaluated (*ConstantNode* (*v* :: *Value*)) = *False*

is-preevaluated (*DynamicNewArrayNode* (*v* :: *nat*) (*va* :: *nat*) (*vb* :: *nat option*) (*vc* :: *nat option*) (*vd* :: *nat*)) = *False*

is-preevaluated EndNode = *False*

is-preevaluated (*ExceptionObjectNode* (*v* :: *nat option*) (*va* :: *nat*)) = *False*

is-preevaluated (*FrameState* (*v* :: *nat list*) (*va* :: *nat option*) (*vb* :: *nat list option*) (*vc* :: *nat list option*)) = *False*

is-preevaluated (*IfNode* (*v* :: *nat*) (*va* :: *nat*) (*vb* :: *nat*)) = *False*

is-preevaluated (*IntegerBelowNode* (*v* :: *nat*) (*va* :: *nat*)) = *False*

is-preevaluated (*IntegerEqualsNode* (*v* :: *nat*) (*va* :: *nat*)) = *False*

is-preevaluated (*IntegerLessThanNode* (*v* :: *nat*) (*va* :: *nat*)) = *False*

is-preevaluated (*IsNullNode* (*v* :: *nat*)) = *False*

is-preevaluated (*KillingBeginNode* (*v* :: *nat*)) = *False*

is-preevaluated (*LeftShiftNode* (*v* :: *nat*) (*va* :: *nat*)) = *False*

is-preevaluated (*LogicNegationNode* (*v* :: *nat*)) = *False*

is-preevaluated (*LoopBeginNode* (*v* :: *nat list*) (*va* :: *nat option*) (*vb* :: *nat option*) (*vc* :: *nat*)) = *False*

is-preevaluated (*LoopEndNode* (*v* :: *nat*)) = *False*

is-preevaluated (*LoopExitNode* (*v* :: *nat*) (*va* :: *nat option*) (*vb* :: *nat*)) = *False*

is-preevaluated (*MergeNode* (*v* :: *nat list*) (*va* :: *nat option*) (*vb* :: *nat*)) = *False*

deterministic-representation

$g :: IRGraph \vdash n :: nat \simeq e_1 :: IRExpr \wedge g \vdash n \simeq e_2 :: IRExpr \Longrightarrow e_1 = e_2$

thm-oracles *repDet*

well-formed-term-graph

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

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-semantics-deterministic

$[g :: IRGraph, m :: nat \Rightarrow Value, p :: Value\ list] \vdash n :: nat \mapsto v_1 :: Value \wedge [g,m,p] \vdash n \mapsto v_2 :: Value \Longrightarrow v_1 = v_2$

thm-oracles *graphDet*

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

graph-refinement

term-graph-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

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

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

maximal-sharing

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

tree-to-graph-rewriting

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

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

term-graph-refines-term

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

term-graph-evaluation

$g :: IRGraph \vdash n :: nat \trianglelefteq e :: IRExpr \Longrightarrow$
$\forall (m{::}nat \Rightarrow Value)\ (p{::}Value\ list)\ v :: Value.$
$[m,p] \vdash e \mapsto v \longrightarrow [g,m,p] \vdash n \mapsto v$

graph-construction

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

thm-oracles *graph-construction*

term-graph-reconstruction

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

refined-insert

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

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

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

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

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

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

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-refinement g1 g2*

maximal-sharing

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

tree-to-graph-rewriting

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

end