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*)

lemma *diff-diff-cancel-expr*:
shows *exp*[e_1 − (e_1 − e_2)] ≥ *exp*[e_2]
apply *simp* **apply** ((*rule allI*)+; *rule impI*)
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

algebraic-laws-expressions

$$e - e \longmapsto \mathit{0} \tag{5}$$
$$e_1 - (e_1 - e_2) \longmapsto e_2 \tag{6}$$

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*:
  $(e - e) \longmapsto ConstantExpr\ (IntVal\ b\ 0)$
    when $((stamp\text{-}expr\ exp[e - e] = IntegerStamp\ b\ lo\ hi) \wedge wf\text{-}stamp\ exp[e - e])$

  apply (*rule impI*) **apply** *simp*
proof −
  assume *assms*: $stamp\text{-}binary\ BinSub\ (stamp\text{-}expr\ e)\ (stamp\text{-}expr\ e) = IntegerStamp\ b\ lo\ hi \wedge wf\text{-}stamp\ exp[e - e]$
  have $\forall m\ p\ v\ .\ ([m,\ p] \vdash exp[e - e] \mapsto v) \longrightarrow (\exists vv.\ v = IntVal\ b\ vv)$
    using *assms wf-stamp-eval*
    by (*metis stamp-expr.simps*(*2*))
  then show $\forall m\ p\ v.\ ([m,p] \vdash BinaryExpr\ BinSub\ e\ e \mapsto v) \longrightarrow ([m,p] \vdash ConstantExpr\ (IntVal\ b\ 0) \mapsto v)$
   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. $\mathit{BinaryExpr}\ \mathit{BinAdd}\ (\mathit{BinaryExpr}\ \mathit{BinSub}\ e_1\ e_2)\ e_2 \sqsupseteq e_1$

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

InverseRightSub

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

InverseRightSubObligation

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

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

expression-refinement-monotone

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

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

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

phase *SnipPhase*
terminating *size*
begin

BinaryFoldConstant

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

BinaryFoldConstantObligation

1. $\mathit{Suc}\ 0 < \mathit{trm}(\mathit{BinaryExpr}\ op\ (\mathit{ConstantExpr}\ v1)\ (\mathit{ConstantExpr}\ v2))$
2. $\mathit{BinaryExpr}\ op\ (\mathit{ConstantExpr}\ v1)\ (\mathit{ConstantExpr}\ v2) \sqsupseteq$
   $\mathit{ConstantExpr}\ (\mathit{bin\text{-}eval}\ op\ v1\ v2)$

using *size-non-const* **apply** *auto*[*1*]
using *BinaryFoldConstant*(*1*) **by** *auto*

AddCommuteConstantRight

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

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

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*

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-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

termination

$trm(UnaryExpr\ op\ e) = trm(e) + 1$

$trm(BinaryExpr\ BinAdd\ x\ y) = trm(x) + trm(y) * 2$

$trm(BinaryExpr\ BinXor\ x\ y) = trm(x) + trm(y)$

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

$trm(ConstantExpr\ c) = 1$

$trm(ParameterExpr\ ind\ s) = 2$

graph-representation

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

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

graph2tree

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

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

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

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

translations
n <= *CONST as-set n*

graph-semantics-preservation

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

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

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

tree-to-graph-rewriting

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

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

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

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

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

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

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

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

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

tree-to-graph-rewriting

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

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