

# Veriopt Theories

September 16, 2022

## Contents

<b>1</b>	<b>Verifying term graph optimizations using Isabelle/HOL</b>	<b>1</b>
1.1	Markup syntax for common operations . . . . .	1
1.2	Representing canonicalization optimizations . . . . .	2
1.3	Representing terms . . . . .	4
1.4	Term semantics . . . . .	5

## 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*  $(- \text{ ? } - : -)$

### 1.1 Markup syntax for common operations

**notation** (*latex*)

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

**notation** (*latex*)

*valid-value*  $(- \in -)$

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

*algebraic-laws*

$$x - x = 0 \quad (1)$$

$$x - (x - y) = y \quad (2)$$

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

*algebraic-laws-values*

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

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

**translations**

$n \leq \text{CONST ConstantExpr } (\text{CONST IntVal } b\ n)$   
 $x - y \leq \text{CONST BinaryExpr } (\text{CONST BinSub})\ x\ y$

**notation** (*ExprRule output*)

*Refines* ( $- \mapsto -$ )

**lemma** *diff-self-expr*:  
**assumes**  $\forall m\ p\ v. [m, p] \vdash \text{exp}[e - e] \mapsto \text{IntVal } b\ v$   
**shows**  $\text{exp}[e - e] \geq \text{exp}[\text{const } (\text{IntVal } b\ 0)]$   
**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  $\exp[e_1 - (e_1 - e_2)] \geq \exp[e_2]$ 
  apply open-eval
  subgoal premises eval for m p v
  proof -
    obtain v1 where v1:  $[m, p] \vdash e_1 \mapsto v1$ 
    using eval by blast
    obtain v2 where v2:  $[m, p] \vdash e_2 \mapsto v2$ 
    using eval by blast
    then have e:  $[m, p] \vdash \exp[e_1 - (e_1 - e_2)] \mapsto \text{val}[v1 - (v1 - v2)]$ 
    using v1 v2 eval
    by (smt (verit, ccfv-SIG) bin-eval.simps(3) evalDet unfold-binary)
    then have notUn:  $\text{val}[v1 - (v1 - v2)] \neq \text{UndefVal}$ 
    using evaltree-not-undef by auto
    then have val:  $\text{val}[v1 - (v1 - v2)] = v2$ 
    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*

$$e - e \mapsto 0 \quad (5)$$

$$e_1 - (e_1 - e_2) \mapsto e_2 \quad (6)$$

**no-translations**

```

n <= CONST ConstantExpr (CONST IntVal b n)
x - y <= CONST BinaryExpr (CONST BinSub) x y

```

**definition** *wf-stamp* :: *IRExpr*  $\Rightarrow$  *bool* **where**

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

**lemma** *wf-stamp-eval*:

```

assumes wf-stamp e
assumes stamp-expr e = IntegerStamp b lo hi
shows  $\forall m \ p \ v. ([m, p] \vdash e \mapsto v) \longrightarrow (\exists vv. v = \text{IntVal } b \ vv)$ 
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) = Inte-
  gerStamp 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] ⊢ Con-
  stantExpr (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 IExpr =  
  UnaryExpr IRUnaryOp IExpr  
| BinaryExpr IRBinaryOp IExpr IExpr  
| ConditionalExpr IExpr IExpr IExpr  
| 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**

$(prop) P \wedge Q \implies R \leq (prop) P \implies Q \implies R$

**translations**

$(prop) P \implies Q \implies R \leq (prop) P \wedge Q \implies R$

*tree-semantics*

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

**no-translations**

$(prop) P \implies Q \implies R \leq (prop) P \wedge Q \implies R$

**translations**

$(prop) P \wedge Q \implies R \leq (prop) P \implies Q \implies R$

And show that expression evaluation is deterministic.

*tree-evaluation-deterministic*

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

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

*expression-refinement*

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

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

**phase** *SnipPhase*  
**terminating** *size*  
**begin**

*InverseLeftSub*

**optimization** *InverseLeftSub*:  
 $(e_1 - e_2) + e_2 \mapsto e_1$

*InverseLeftSubObligation*

1. *BinaryExpr BinAdd (BinaryExpr BinSub e<sub>1</sub> e<sub>2</sub>) e<sub>2</sub>  $\sqsubseteq$  e<sub>1</sub>*

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

*InverseRightSub*

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

*InverseRightSubObligation*

1. *BinaryExpr BinAdd e<sub>2</sub> (BinaryExpr BinSub e<sub>1</sub> e<sub>2</sub>)  $\sqsubseteq$  e<sub>1</sub>*

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

*expression-refinement-monotone*

$e \sqsupseteq e' \implies \text{UnaryExpr } op \ e \sqsupseteq \text{UnaryExpr } op \ e'$

$x \sqsupseteq x' \wedge y \sqsupseteq y' \implies \text{BinaryExpr } op \ x \ y \sqsupseteq \text{BinaryExpr } op \ x' \ y'$

$ce \sqsupseteq ce' \wedge te \sqsupseteq te' \wedge fe \sqsupseteq fe' \implies$   
 $\text{ConditionalExpr } ce \ te \ fe \sqsupseteq \text{ConditionalExpr } ce' \ te' \ fe'$

**phase** *SnipPhase*  
**terminating** *size*  
**begin**

*BinaryFoldConstant*

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

*BinaryFoldConstantObligation*

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

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

*AddCommuteConstantRight*

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

*AddCommuteConstantRightObligation*

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

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

*AddNeutral*

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

*AddNeutralObligation*

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

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

*AddToSub*

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

*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*:  $((!e) ? x : y) \mapsto (e ? y : x)$  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*:  $(\text{true} \ ? \ x : y) \mapsto x$

**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*:  $(e \ ? \ x : x) \mapsto x$

**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*)

**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*:  $((x < y) \ ? \ x : y) \mapsto y$   
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*

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

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

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

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

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

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

#### *graph-representation*

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

#### **no-translations**

$$(prop) \ P \wedge Q \implies R \leq (prop) \ P \implies Q \implies R$$

#### **translations**

$$(prop) \ P \implies Q \implies R \leq (prop) \ P \wedge Q \implies R$$

#### *graph2tree*

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

#### **no-translations**

$$(prop) \ P \implies Q \implies R \leq (prop) \ P \wedge Q \implies R$$

#### **translations**

$$(prop) \ P \wedge Q \implies R \leq (prop) \ P \implies Q \implies R$$

## *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*

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

**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*

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

**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-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*

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

### *graph-refinement*

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

### **translations**

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

### *graph-semantics-preservation*

$$\begin{aligned} &\llbracket (e1'::IRExpr) \sqsupseteq \\ &\quad (e2'::IRExpr); \\ &\quad \{n'::nat\} \triangleleft g1::IRGraph \\ &\quad \subseteq (g2::IRGraph); \\ &\quad g1 \vdash n' \simeq e1'; g2 \vdash n' \simeq e2' \rrbracket \\ &\implies graph\text{-}refinement\ g1\ g2 \end{aligned}$$

### *maximal-sharing*

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

*tree-to-graph-rewriting*

$$\begin{aligned} & (e_1::IRExpr) \sqsupseteq (e_2::IRExpr) \wedge \\ & g_1::IRGraph \vdash n::nat \simeq e_1 \wedge \\ & \text{maximal-sharing } g_1 \wedge \\ & \{n\} \triangleleft g_1 \subseteq (g_2::IRGraph) \wedge \\ & g_2 \vdash n \simeq e_2 \wedge \text{maximal-sharing } g_2 \implies \\ & \text{graph-refinement } g_1 \ g_2 \end{aligned}$$

*graph-represents-expression*

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

*graph-construction*

$$\begin{aligned} & (e_1::IRExpr) \sqsupseteq (e_2::IRExpr) \wedge \\ & (g_1::IRGraph) \subseteq (g_2::IRGraph) \wedge \\ & g_2 \vdash n::nat \simeq e_2 \implies \\ & g_2 \vdash n \leq e_1 \wedge \text{graph-refinement } g_1 \ g_2 \end{aligned}$$

end