

# 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**  
     $y > x \leq x < y$   
  
**notation** (*latex*)  
    *greater* (- > / -)

**translations**

$n \leq \text{CONST Rep-intexp } n$   
 $n \leq \text{CONST Rep-i32exp } n$

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

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

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

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

*val-eq*

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

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

**lemma** *sub-same-val*:  
**assumes**  $\text{val}[e - e] = \text{IntVal } b \ v$   
**shows**  $\text{val}[e - e] = \text{val}[\text{IntVal } b \ 0]$   
**using** *assms* **by** (*cases e; auto*)

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

**lemma** *wf-stamp-eval*:  
**assumes** *wf-stamp e*  
**assumes**  $\text{stamp-expr } e = \text{IntegerStamp } b \ lo \ hi$   
**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** *tmp*  
**terminating** *size*  
**begin**

*sub-same-32*

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

**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*  $\wedge$  *wf-stamp* *exp*[*e* – *e*]  
**have**  $\forall m \ p \ v. ([m, p] \vdash \text{exp}[e - e] \mapsto v) \longrightarrow (\exists vv. v = \text{IntVal } b \ vv)$   
**using** *assms wf-stamp-eval*  
**by** (*metis stamp-expr.simps(2)*)  
**then show**  $\forall m \ p \ v. ([m, p] \vdash \text{BinaryExpr BinSub } e \ e \mapsto v) \longrightarrow ([m, p] \vdash \text{ConstantExpr } (\text{IntVal } b \ 0) \mapsto v)$   
**by** (*smt (verit, best) BinaryExprE Tree.Snippets.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-antics*

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

*tree-evaluation-deterministic*

```
[m :: nat ⇒ Value, p :: Value list] ⊢ e :: IRExp ↦ v1 :: Value ∧
[m, p] ⊢ e ↦ v2 :: Value ⇒
v1 = v2
```

**thm-oracles** *evalDet*

*expression-refinement*

```
(e1 :: IRExp) ⊑ (e2 :: IRExp) = (∀ (m :: nat ⇒ Value) (p :: Value list)
v :: Value. [m, p] ⊢ e1 ↦ v → [m, p] ⊢ e2 ↦ v)
```

*expression-refinement-monotone*

```
(e :: IRExp) ⊑ (e' :: IRExp)
(x :: IRExp) ⊑ (x' :: IRExp) ∧ (y :: IRExp) ⊑ (y' :: IRExp)
(ce :: IRExp) ⊑ (ce' :: IRExp) ∧ (te :: IRExp) ⊑ (te' :: IRExp) ∧ (fe :: IRExp) ⊑ (fe' :: IRExp)
```

**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*)  
 $\mapsto$  *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*  $v1\ v2 \longrightarrow$   
     $BinaryExpr\ op\ (ConstantExpr\ v1)\ (ConstantExpr\ v2) \sqsubseteq$   
     $ConstantExpr\ (bin-eval\ op\ v1\ v2)$   
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.  $\neg is-ConstantExpr\ y \longrightarrow$   
     $BinaryExpr\ BinAdd\ (ConstantExpr\ v)\ y \sqsubseteq$   
     $BinaryExpr\ BinAdd\ y\ (ConstantExpr\ v)$   
variables:  
     $v :: Value$   
     $y :: IRExp$

**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*)))  
⊑  
  *e*  
variables:  
  *e* :: *i32exp*

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

#### *InverseLeftSub*

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

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

#### *InverseLeftSubObligation*

1. *trm*(*e*<sub>2</sub>) > 0 :: *nat*  
2. *BinaryExpr* *BinAdd* (*BinaryExpr* *BinSub* *e*<sub>1</sub> *e*<sub>2</sub>) *e*<sub>2</sub> ⊑ *e*<sub>1</sub>  
variables:  
  *e*<sub>1</sub>, *e*<sub>2</sub> :: *intexp*

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

#### *InverseRightSub*

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

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

#### *InverseRightSubObligation*

1. *trm*(*e*<sub>1</sub>) > 0 :: *nat* ∨ *trm*(*e*<sub>2</sub>) > 0 :: *nat*  
2. *BinaryExpr* *BinAdd* *e*<sub>2</sub> (*BinaryExpr* *BinSub* *e*<sub>1</sub> *e*<sub>2</sub>) ⊑ *e*<sub>1</sub>  
variables:  
  *e*<sub>1</sub>, *e*<sub>2</sub> :: *intexp*

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

#### *AddToSub*

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

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

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

*AddToSubObligation*

1. *BinaryExpr BinAdd (UnaryExpr UnaryNeg e) y*  $\sqsubseteq$  *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*:  $((!e) \text{ ? } x : y) \mapsto (e \text{ ? } y : x)$

**using** *ConditionalPhase.NegateConditionFlipBranches*

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

*phase-example-2*

**optimization** *const-true*:  $(\text{true} \text{ ? } x : y) \mapsto x$

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

*phase-example-3*

**optimization** *const-false*:  $(\text{false} \text{ ? } x : y) \mapsto y$

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



*phase-example-4*

**optimization** *equal-branches*:  $(e \text{ ? } x : x) \mapsto x$

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

*phase-example-7*

**end**

*termination*

$trm(UnaryExpr \ (op :: IRUnaryOp) \ (e :: IRExp))$	$= \ trm(e :: IRExp) + (1 :: nat)$
$trm(BinaryExpr \ BinAdd \ (x :: IRExp) \ (y :: IRExp))$	$= \ trm(x :: IRExp) + (2 :: nat)$
$trm(BinaryExpr \ BinIntegerBelow \ (x :: IRExp) \ (y :: IRExp))$	$= \ trm(x :: IRExp) + trm(y :: IRExp)$
$trm(ConditionalExpr \ (cond :: IRExp) \ (t :: IRExp) \ (f :: IRExp))$	$= \ trm(cond :: IRExp) + \max(trm(t :: IRExp), trm(f :: IRExp))$
$trm(ConstantExpr \ (c :: Value))$	$= \ 1 :: nat$
$trm(ParameterExpr \ (ind :: nat) \ (s :: Stamp))$	$= \ 2 :: nat$

*graph-representation*

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

*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* (*h0* :: *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*

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

**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*

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

**thm-oracles** *graphDet*

**notation** (*latex*)

*graph-refinement* (*term-graph-refinement* -)

*graph-refinement*

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

**translations**

$n \leq CONST$  as-set  $n$

*graph-antics-preservation*

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

**thm-oracles** *graph-antics-preservation-subscript*

*maximal-sharing*

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

*tree-to-graph-rewriting*

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

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

*term-graph-refines-term*

$$\begin{aligned}
& (g :: IRGraph \vdash n :: nat \trianglelefteq e :: IRExp) = \\
& (\exists e' :: IRExp. g \vdash n \simeq e' \wedge e \sqsupseteq e')
\end{aligned}$$

*term-graph-evaluation*

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

*graph-construction*

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

**thm-oracles** *graph-construction*

*term-graph-reconstruction*

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

*refined-insert*

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

**end**

**theory** *SlideSnippets*

**imports**

*Semantics.TreeToGraphThms*

*Snippets.Snipping*

**begin**

**notation** (*latex*)

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

**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*

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

#### *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*

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

#### **translations**

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

*graph-antics-preservation*

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

*maximal-sharing*

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

*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*

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

*graph-construction*

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

end