# Veriopt Theories

August 31, 2022

## Contents



# 1 Data-flow Semantics

**theory** *IRTreeEval*
**imports**
*Graph.Stamp*
**begin**

We define a tree representation of data-flow nodes, as an abstraction of the graph view.

Data-flow trees are evaluated in the context of a method state (currently called MapState in the theories for historical reasons).

The method state consists of the values for each method parameter, references to method parameters use an index of the parameter within the parameter list, as such we store a list of parameter values which are looked up at parameter references.

The method state also stores a mapping of node ids to values. The contents of this mapping is calculates during the traversal of the control flow graph.

As a concrete example, as the *SignedDivNode* can have side-effects (during division by zero), it is treated as part of the control-flow, since the data-flow phase is specified to be side-effect free. As a result, the control-flow semantics for *SignedDivNode* calculates the value of a node and maps the node identifier to the value within the method state. The data-flow semantics then just reads the value stored in the method state for the node.

**type-synonym** *ID* = *nat*
**type-synonym** *MapState* = *ID* ⇒ *Value*
**type-synonym** *Params* = *Value list*

**definition** *new-map-state* :: *MapState* **where**
*new-map-state* = (*λx. UndefVal*)

## 1.1 Data-flow Tree Representation

**datatype** *IRUnaryOp* =
*UnaryAbs*
| *UnaryNeg*
| *UnaryNot*
| *UnaryLogicNegation*
| *UnaryNarrow* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*)
| *UnarySignExtend* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*)
| *UnaryZeroExtend* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*)

**datatype** *IRBinaryOp* =
*BinAdd*
| *BinMul*
| *BinSub*
| *BinAnd*
| *BinOr*
| *BinXor*
| *BinShortCircuitOr*
| *BinLeftShift*
| *BinRightShift*
| *BinURightShift*
| *BinIntegerEquals*
| *BinIntegerLessThan*
| *BinIntegerBelow*

**datatype** (*discs-sels*) *IRExpr* =
*UnaryExpr* (*ir-uop*: *IRUnaryOp*) (*ir-value*: *IRExpr*)

| *BinaryExpr* (*ir-op*: *IRBinaryOp*) (*ir-x*: *IRExpr*) (*ir-y*: *IRExpr*)
| *ConditionalExpr* (*ir-condition*: *IRExpr*) (*ir-trueValue*: *IRExpr*) (*ir-falseValue*: *IRExpr*)

| *ParameterExpr* (*ir-index*: *nat*) (*ir-stamp*: *Stamp*)

| *LeafExpr* (*ir-nid*: *ID*) (*ir-stamp*: *Stamp*)

| *ConstantExpr* (*ir-const*: *Value*)
| *ConstantVar* (*ir-name*: *string*)
| *VariableExpr* (*ir-name*: *string*) (*ir-stamp*: *Stamp*)

**fun** *is-ground* :: *IRExpr* ⇒ *bool* **where**
*is-ground* (*UnaryExpr op e*) = *is-ground e* |
*is-ground* (*BinaryExpr op e1 e2*) = (*is-ground e1* ∧ *is-ground e2*) |
*is-ground* (*ConditionalExpr b e1 e2*) = (*is-ground b* ∧ *is-ground e1* ∧ *is-ground e2*) |
*is-ground* (*ParameterExpr i s*) = *True* |
*is-ground* (*LeafExpr n s*) = *True* |
*is-ground* (*ConstantExpr v*) = *True* |
*is-ground* (*ConstantVar name*) = *False* |
*is-ground* (*VariableExpr name s*) = *False*

**typedef** *GroundExpr* = { *e* :: *IRExpr* . *is-ground e* }
**using** *is-ground.simps*(*6*) **by** *blast*

## 1.2 Functions for re-calculating stamps

Note: in Java all integer calculations are done as 32 or 64 bit calculations. However, here we generalise the operators to allow any size calculations. Most operators have the same output bits as their inputs. But the following $fixed_32$ binary operators always output 32 bits. And the unary operators that are not $normal_unary$ are narrowing or widening operators, so the result bits is specified by the operator.

**abbreviation** *fixed-32* :: *IRBinaryOp set* **where**
*fixed-32* ≡ {*BinShortCircuitOr*, *BinIntegerEquals*, *BinIntegerLessThan*, *BinIntegerBelow*}

**abbreviation** *normal-unary* :: *IRUnaryOp set* **where**
*normal-unary* ≡ {*UnaryAbs*, *UnaryNeg*, *UnaryNot*, *UnaryLogicNegation*}

**fun** *stamp-unary* :: *IRUnaryOp* ⇒ *Stamp* ⇒ *Stamp* **where**

*stamp-unary op* (*IntegerStamp b lo hi*) =
*unrestricted-stamp* (*IntegerStamp* (*if op* ∈ *normal-unary then b else* (*ir-resultBits op*)) *lo hi*) |

*stamp-unary op - = IllegalStamp*

**fun** *stamp-binary* :: *IRBinaryOp* ⇒ *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
*stamp-binary op* (*IntegerStamp b1 lo1 hi1*) (*IntegerStamp b2 lo2 hi2*) =
(*if* (*b1* ≠ *b2*) *then IllegalStamp else*
(*if op* ∈ *fixed-32*
*then unrestricted-stamp* (*IntegerStamp 32 lo1 hi1*)
*else unrestricted-stamp* (*IntegerStamp b1 lo1 hi1*))) |

*stamp-binary op - - = IllegalStamp*

**fun** *stamp-expr* :: *IRExpr* ⇒ *Stamp* **where**
*stamp-expr* (*UnaryExpr op x*) = *stamp-unary op* (*stamp-expr x*) |
*stamp-expr* (*BinaryExpr bop x y*) = *stamp-binary bop* (*stamp-expr x*) (*stamp-expr y*) |
*stamp-expr* (*ConstantExpr val*) = *constantAsStamp val* |
*stamp-expr* (*LeafExpr i s*) = *s* |
*stamp-expr* (*ParameterExpr i s*) = *s* |
*stamp-expr* (*ConditionalExpr c t f*) = *meet* (*stamp-expr t*) (*stamp-expr f*)

**export-code** *stamp-unary stamp-binary stamp-expr*

## 1.3 Data-flow Tree Evaluation

**fun** *unary-eval* :: *IRUnaryOp* ⇒ *Value* ⇒ *Value* **where**
*unary-eval UnaryAbs v* = *intval-abs v* |
*unary-eval UnaryNeg v* = *intval-negate v* |
*unary-eval UnaryNot v* = *intval-not v* |
*unary-eval UnaryLogicNegation v* = *intval-logic-negation v* |
*unary-eval* (*UnaryNarrow inBits outBits*) *v* = *intval-narrow inBits outBits v* |
*unary-eval* (*UnarySignExtend inBits outBits*) *v* = *intval-sign-extend inBits outBits v* |
*unary-eval* (*UnaryZeroExtend inBits outBits*) *v* = *intval-zero-extend inBits outBits v*

**fun** *bin-eval* :: *IRBinaryOp* ⇒ *Value* ⇒ *Value* ⇒ *Value* **where**
*bin-eval BinAdd v1 v2* = *intval-add v1 v2* |
*bin-eval BinMul v1 v2* = *intval-mul v1 v2* |
*bin-eval BinSub v1 v2* = *intval-sub v1 v2* |
*bin-eval BinAnd v1 v2* = *intval-and v1 v2* |
*bin-eval BinOr v1 v2* = *intval-or v1 v2* |
*bin-eval BinXor v1 v2* = *intval-xor v1 v2* |
*bin-eval BinShortCircuitOr v1 v2* = *intval-short-circuit-or v1 v2* |
*bin-eval BinLeftShift v1 v2* = *intval-left-shift v1 v2* |
*bin-eval BinRightShift v1 v2* = *intval-right-shift v1 v2* |
*bin-eval BinURightShift v1 v2* = *intval-uright-shift v1 v2* |
*bin-eval BinIntegerEquals v1 v2* = *intval-equals v1 v2* |
*bin-eval BinIntegerLessThan v1 v2* = *intval-less-than v1 v2* |

*bin-eval BinIntegerBelow v1 v2* = *intval-below v1 v2*

**lemmas** *eval-thms* =
*intval-abs.simps intval-negate.simps intval-not.simps*
*intval-logic-negation.simps intval-narrow.simps*
*intval-sign-extend.simps intval-zero-extend.simps*
*intval-add.simps intval-mul.simps intval-sub.simps*
*intval-and.simps intval-or.simps intval-xor.simps*
*intval-left-shift.simps intval-right-shift.simps*
*intval-uright-shift.simps intval-equals.simps*
*intval-less-than.simps intval-below.simps*

**inductive** *not-undef-or-fail* :: *Value* ⇒ *Value* ⇒ *bool* **where**
⟦*value* ≠ *UndefVal*⟧ ⟹ *not-undef-or-fail value value*

**notation** (*latex* **output**)
*not-undef-or-fail* (- = -)

**inductive**
*evaltree* :: *MapState* ⇒ *Params* ⇒ *IRExpr* ⇒ *Value* ⇒ *bool* ([-,-] ⊢ - ↦ - *55*)
**for** *m p* **where**

*ConstantExpr*:
⟦*valid-value c* (*constantAsStamp c*)⟧
⟹ [*m*,*p*] ⊢ (*ConstantExpr c*) ↦ *c* |

*ParameterExpr*:
⟦*i* < *length p*; *valid-value* (*p*!*i*) *s*⟧
⟹ [*m*,*p*] ⊢ (*ParameterExpr i s*) ↦ *p*!*i* |

*ConditionalExpr*:
⟦[*m*,*p*] ⊢ *ce* ↦ *cond*;
*branch* = (*if val-to-bool cond then te else fe*);
[*m*,*p*] ⊢ *branch* ↦ *v*;
*v* ≠ *UndefVal*⟧
⟹ [*m*,*p*] ⊢ (*ConditionalExpr ce te fe*) ↦ *v* |

*UnaryExpr*:
⟦[*m*,*p*] ⊢ *xe* ↦ *v*;
*result* = (*unary-eval op v*);
*result* ≠ *UndefVal*⟧
⟹ [*m*,*p*] ⊢ (*UnaryExpr op xe*) ↦ *result* |

*BinaryExpr*:
⟦[*m*,*p*] ⊢ *xe* ↦ *x*;
[*m*,*p*] ⊢ *ye* ↦ *y*;
*result* = (*bin-eval op x y*);

*result* $\neq$ *UndefVal*⟧
  $\Longrightarrow$ [*m*,*p*] $\vdash$ (*BinaryExpr op xe ye*) $\mapsto$ *result* |

*LeafExpr*:
⟦*val* = *m n*;
  *valid-value val s*⟧
  $\Longrightarrow$ [*m*,*p*] $\vdash$ *LeafExpr n s* $\mapsto$ *val*

**code-pred** (*modes*: *i* $\Rightarrow$ *i* $\Rightarrow$ *i* $\Rightarrow$ *o* $\Rightarrow$ *bool as evalT*)
  [*show-steps*,*show-mode-inference*,*show-intermediate-results*]
  *evaltree* **.**

**inductive**
  *evaltrees* :: *MapState* $\Rightarrow$ *Params* $\Rightarrow$ *IRExpr list* $\Rightarrow$ *Value list* $\Rightarrow$ *bool* ([-,-] $\vdash$ - $\mapsto_L$ - *55*)
  **for** *m p* **where**

*EvalNil*:
[*m*,*p*] $\vdash$ [] $\mapsto_L$ [] |

*EvalCons*:
⟦[*m*,*p*] $\vdash$ *x* $\mapsto$ *xval*;
  [*m*,*p*] $\vdash$ *yy* $\mapsto_L$ *yyval*⟧
  $\Longrightarrow$ [*m*,*p*] $\vdash$ (*x*#*yy*) $\mapsto_L$ (*xval*#*yyval*)

**code-pred** (*modes*: *i* $\Rightarrow$ *i* $\Rightarrow$ *i* $\Rightarrow$ *o* $\Rightarrow$ *bool as evalTs*)
  *evaltrees* **.**

**definition** *sq-param0* :: *IRExpr* **where**
  *sq-param0* = *BinaryExpr BinMul*
    (*ParameterExpr 0* (*IntegerStamp 32* (− *2147483648*) *2147483647*))
    (*ParameterExpr 0* (*IntegerStamp 32* (− *2147483648*) *2147483647*))

**values** {*v*. *evaltree new-map-state* [*IntVal 32 5*] *sq-param0 v*}

**declare** *evaltree.intros* [*intro*]
**declare** *evaltrees.intros* [*intro*]

## 1.4 Data-flow Tree Refinement

We define the induced semantic equivalence relation between expressions. Note that syntactic equality implies semantic equivalence, but not vice versa.

**definition** *equiv-exprs* :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* (- $\doteq$ - *55*) **where**
  (*e1* $\doteq$ *e2*) = ($\forall$ *m p v*. (([*m*,*p*] $\vdash$ *e1* $\mapsto$ *v*) $\longleftrightarrow$ ([*m*,*p*] $\vdash$ *e2* $\mapsto$ *v*)))

We also prove that this is a total equivalence relation (*equivp equiv-exprs*)

(HOL.Equiv_Relations), so that we can reuse standard results about equivalence relations.

**lemma** *equivp equiv-exprs*
  **apply** (*auto simp add*: *equivp-def equiv-exprs-def*)
  **by** (*metis equiv-exprs-def*)+

We define a refinement ordering over IRExpr and show that it is a preorder. Note that it is asymmetric because e2 may refer to fewer variables than e1.

**instantiation** *IRExpr* :: *preorder* **begin**

**notation** *less-eq* (**infix** $\sqsubseteq$ *65*)

**definition**
  *le-expr-def* [*simp*]:
    $(e_2 \leq e_1) \longleftrightarrow (\forall\ m\ p\ v.\ (([m,p] \vdash e_1 \mapsto v) \longrightarrow ([m,p] \vdash e_2 \mapsto v)))$

**definition**
  *lt-expr-def* [*simp*]:
    $(e_1 < e_2) \longleftrightarrow (e_1 \leq e_2 \wedge \neg\ (e_1 \doteq e_2))$

**instance proof**
  **fix** $x\ y\ z$ :: *IRExpr*
  **show** $x < y \longleftrightarrow x \leq y \wedge \neg\ (y \leq x)$ **by** (*simp add*: *equiv-exprs-def*; *auto*)
  **show** $x \leq x$ **by** *simp*
  **show** $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$ **by** *simp*
**qed**

**end**

**abbreviation** (**output**) *Refines* :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* (**infix** $\sqsupseteq$ *64*)
  **where** $e_1 \sqsupseteq e_2 \equiv (e_2 \leq e_1)$

**end**

## 1.5 Data-flow Tree Theorems

**theory** *IRTreeEvalThms*
  **imports**
    *Graph.ValueThms*
    *IRTreeEval*
**begin**

### 1.5.1 Deterministic Data-flow Evaluation

**lemma** *evalDet*:
  $[m,p] \vdash e \mapsto v_1 \Longrightarrow$
   $[m,p] \vdash e \mapsto v_2 \Longrightarrow$
  $v_1 = v_2$
  **apply** (*induction arbitrary*: $v_2$ *rule*: *evaltree.induct*)

**by** (*elim EvalTreeE*; *auto*)+

**lemma** *evalAllDet*:
  $[m,p] \vdash e \mapsto_L v1 \Longrightarrow$
   $[m,p] \vdash e \mapsto_L v2 \Longrightarrow$
   $v1 = v2$
  **apply** (*induction arbitrary*: *v2 rule*: *evaltrees.induct*)
   **apply** (*elim EvalTreeE*; *auto*)
  **using** *evalDet* **by** *force*

### 1.5.2 Typing Properties for Integer Evaluation Functions

We use three simple typing properties on integer values: $is_IntVal32, is_IntVal64$ and the more general $is_IntVal$.

**lemma** *unary-eval-not-obj-ref*:
  **shows** *unary-eval op x* $\neq$ *ObjRef v*
  **by** (*cases op*; *cases x*; *auto*)

**lemma** *unary-eval-not-obj-str*:
  **shows** *unary-eval op x* $\neq$ *ObjStr v*
  **by** (*cases op*; *cases x*; *auto*)

**lemma** *unary-eval-int*:
  **assumes** *def*: *unary-eval op x* $\neq$ *UndefVal*
  **shows** *is-IntVal* (*unary-eval op x*)
  **unfolding** *is-IntVal-def* **using** *def*
  **apply** (*cases unary-eval op x*; *auto*)
  **using** *unary-eval-not-obj-ref unary-eval-not-obj-str* **by** *simp*+

**lemma** *bin-eval-int*:
  **assumes** *def*: *bin-eval op x y* $\neq$ *UndefVal*
  **shows** *is-IntVal* (*bin-eval op x y*)
  **apply** (*cases op*; *cases x*; *cases y*)
  **unfolding** *is-IntVal-def* **using** *def* **apply** *auto*
                **apply** *presburger*+
          **apply** (*meson bool-to-val.elims*)
         **apply** (*meson bool-to-val.elims*)
        **apply** (*smt* (*verit*) *new-int.simps*)+
  **by** (*meson bool-to-val.elims*)+

**lemma** *IntVal0*:
  (*IntVal 32 0*) = (*new-int 32 0*)
  **unfolding** *new-int.simps*

**by** *auto*

**lemma** *IntVal1*:
($IntVal\ 32\ 1$) = ($new\text{-}int\ 32\ 1$)
**unfolding** *new-int.simps*
**by** *auto*

**lemma** *bin-eval-new-int*:
**assumes** *def*: $bin\text{-}eval\ op\ x\ y \neq UndefVal$
**shows** $\exists\, b\ v.\ (bin\text{-}eval\ op\ x\ y) = new\text{-}int\ b\ v \wedge$
$b = (if\ op \in fixed\text{-}32\ then\ 32\ else\ intval\text{-}bits\ x)$
**apply** (*cases op*; *cases x*; *cases y*)
**unfolding** *is-IntVal-def* **using** *def* **apply** *auto*
**apply** *presburger+*
**apply** (*metis take-bit-and*)
**apply** *presburger*
**apply** (*metis take-bit-or*)
**apply** *presburger*
**apply** (*metis take-bit-xor*)
**apply** *presburger*
**using** *IntVal0 IntVal1*
**apply** (*metis bool-to-val.elims new-int.simps*)
**apply** *presburger*
**apply** (*smt* (*verit*) *new-int.elims*)
**apply** (*smt* (*verit*, *best*) *new-int.elims*)
**apply** (*metis IntVal0 IntVal1 bool-to-val.elims new-int.simps*)
**apply** *presburger*
**apply** (*metis IntVal0 IntVal1 bool-to-val.elims new-int.simps*)
**apply** *presburger*
**apply** (*metis IntVal0 IntVal1 bool-to-val.elims new-int.simps*)
**by** *meson*

**lemma** *int-stamp*:
**assumes** *i*: *is-IntVal v*
**shows** *is-IntegerStamp* (*constantAsStamp v*)
**using** *i* **unfolding** *is-IntegerStamp-def is-IntVal-def* **by** *auto*

**lemma** *validStampIntConst*:
**assumes** $v = IntVal\ b\ ival$
**assumes** $0 < b \wedge b \leq 64$
**shows** *valid-stamp* (*constantAsStamp v*)
**proof** −
**have** *bnds*: $fst\ (bit\text{-}bounds\ b) \leq int\text{-}signed\text{-}value\ b\ ival \wedge int\text{-}signed\text{-}value\ b\ ival \leq snd\ (bit\text{-}bounds\ b)$
**using** *assms int-signed-value-bounds*
**by** *presburger*

**have** *s*: *constantAsStamp v* = *IntegerStamp b* (*int-signed-value b ival*) (*int-signed-value b ival*)
  **using** *assms*(*1*) *constantAsStamp.simps*(*1*) **by** *blast*
 **then show** *?thesis*
  **unfolding** *s valid-stamp.simps*
  **using** *assms*(*2*) *assms bnds* **by** *linarith*
**qed**

**lemma** *validDefIntConst*:
 **assumes** *v*: $v = IntVal\ b\ ival$
 **assumes** $0 < b \wedge b \leq 64$
 **assumes** *take-bit b ival* = *ival*
 **shows** *valid-value v* (*constantAsStamp v*)
**proof** −
 **have** *bnds*: *fst* (*bit-bounds b*) ≤ *int-signed-value b ival* ∧ *int-signed-value b ival* ≤ *snd* (*bit-bounds b*)
  **using** *assms int-signed-value-bounds*
  **by** *presburger*
 **have** *s*: *constantAsStamp v* = *IntegerStamp b* (*int-signed-value b ival*) (*int-signed-value b ival*)
  **using** *assms*(*1*) *constantAsStamp.simps*(*1*) **by** *blast*
 **then show** *?thesis*
  **unfolding** *s* **unfolding** *v* **unfolding** *valid-value.simps*
  **using** *assms validStampIntConst*
  **by** *simp*
**qed**

### 1.5.3 Evaluation Results are Valid

A valid value cannot be $UndefVal$.

**lemma** *valid-not-undef*:
 **assumes** *a1*: *valid-value val s*
 **assumes** *a2*: *s* ≠ *VoidStamp*
 **shows** *val* ≠ *UndefVal*
 **apply** (*rule valid-value.elims*(*1*)[*of val s True*])
 **using** *a1 a2* **by** *auto*

**lemma** *valid-VoidStamp*[*elim*]:
 **shows** *valid-value val VoidStamp* ⟹
   *val* = *UndefVal*
 **using** *valid-value.simps* **by** *metis*

**lemma** *valid-ObjStamp*[*elim*]:
 **shows** *valid-value val* (*ObjectStamp klass exact nonNull alwaysNull*) ⟹
   (∃ *v*. *val* = *ObjRef v*)
 **using** *valid-value.simps* **by** (*metis val-to-bool.cases*)

**lemma** *valid-int*[*elim*]:

**shows** *valid-value val* (*IntegerStamp b lo hi*) $\Longrightarrow$
  ($\exists$ *v*. *val* = *IntVal b v*)
**using** *valid-value.elims*(*2*) **by** *fastforce*

**lemmas** *valid-value-elims* =
*valid-VoidStamp*
*valid-ObjStamp*
*valid-int*

**lemma** *evaltree-not-undef*:
**fixes** *m p e v*
**shows** ([*m*,*p*] $\vdash$ *e* $\mapsto$ *v*) $\Longrightarrow$ *v* $\neq$ *UndefVal*
**apply** (*induction rule*: *evaltree.induct*)
**using** *valid-not-undef* **by** *auto*

**lemma** *leafint*:
**assumes** *ev*: [*m*,*p*] $\vdash$ *LeafExpr i* (*IntegerStamp b lo hi*) $\mapsto$ *val*
**shows** $\exists$ *b v*. *val* = (*IntVal b v*)

**proof** –
**have** *valid-value val* (*IntegerStamp b lo hi*)
  **using** *ev* **by** (*rule LeafExprE*; *simp*)
**then show** *?thesis* **by** *auto*
**qed**

**lemma** *default-stamp* [*simp*]: *default-stamp* = *IntegerStamp 32* (*−2147483648*) *2147483647*
**using** *default-stamp-def* **by** *auto*

**lemma** *valid-value-signed-int-range* [*simp*]:
**assumes** *valid-value val* (*IntegerStamp b lo hi*)
**assumes** *lo* < *0*
**shows** $\exists$ *v*. (*val* = *IntVal b v* $\wedge$
  *lo* $\leq$ *int-signed-value b v* $\wedge$
  *int-signed-value b v* $\leq$ *hi*)
**using** *assms valid-int*
**by** (*metis valid-value.simps*(*1*))

### 1.5.4 Example Data-flow Optimisations

### 1.5.5 Monotonicity of Expression Refinement

We prove that each subexpression position is monotonic. That is, optimizing a subexpression anywhere deep inside a top-level expression also optimizes that top-level expression.

Note that we might also be able to do this via reusing Isabelle's *mono* operator (HOL.Orderings theory), proving instantiations like $mono(UnaryExprop)$, but it is not obvious how to do this for both arguments of the binary expressions.

**lemma** *mono-unary*:
 **assumes** $e \geq e'$
 **shows** $(UnaryExpr\ op\ e) \geq (UnaryExpr\ op\ e')$
 **using** *UnaryExpr assms* **by** *auto*

**lemma** *mono-binary*:
 **assumes** $x \geq x'$
 **assumes** $y \geq y'$
 **shows** $(BinaryExpr\ op\ x\ y) \geq (BinaryExpr\ op\ x'\ y')$
 **using** *BinaryExpr assms* **by** *auto*

**lemma** *never-void*:
 **assumes** $[m,\ p] \vdash x \mapsto xv$
 **assumes** *valid-value xv* (*stamp-expr xe*)
 **shows** *stamp-expr xe* $\neq$ *VoidStamp*
 **using** *valid-value.simps*
 **using** *assms*(*2*) **by** *force*

**lemma** *compatible-trans*:
 *compatible x y* $\wedge$ *compatible y z* $\Longrightarrow$ *compatible x z*
 **by** (*smt* (*z3*) *compatible.elims*(*2*) *compatible.simps*(*1*))

**lemma** *compatible-refl*:
 *compatible x y* $\Longrightarrow$ *compatible y x*
 **using** *compatible.elims*(*2*) **by** *fastforce*

**lemma** *mono-conditional*:
 **assumes** $ce \geq ce'$
 **assumes** $te \geq te'$
 **assumes** $fe \geq fe'$
 **shows** $(ConditionalExpr\ ce\ te\ fe) \geq (ConditionalExpr\ ce'\ te'\ fe')$
**proof** (*simp only*: *le-expr-def*; (*rule allI*)+; *rule impI*)
 **fix** *m p v*
 **assume** *a*: $[m,p] \vdash ConditionalExpr\ ce\ te\ fe \mapsto v$
 **then obtain** *cond* **where** *ce*: $[m,p] \vdash ce \mapsto cond$ **by** *auto*
 **then have** *ce'*: $[m,p] \vdash ce' \mapsto cond$ **using** *assms* **by** *auto*

 **define** *branch* **where** *b*: *branch* = (*if val-to-bool cond then te else fe*)
 **define** *branch'* **where** *b'*: *branch'* = (*if val-to-bool cond then te' else fe'*)
 **then have** *beval*: $[m,p] \vdash branch \mapsto v$ **using** *a b ce evalDet* **by** *blast*

**from** *beval* **have** $[m,p] \vdash branch' \mapsto v$ **using** *assms b b'* **by** *auto*
**then show** $[m,p] \vdash ConditionalExpr\ ce'\ te'\ fe' \mapsto v$
**using** *ConditionalExpr ce' b'*
**using** *a* **by** *blast*
**qed**

## 1.6 Unfolding rules for evaltree quadruples down to bin-eval level

These rewrite rules can be useful when proving optimizations. They support top-down rewriting of each level of the tree into the lower-level $bin_eval$ / $unary_eval$ level, simply by saying $unfoldingunfold_evaltree$.

**lemma** *unfold-const*:
**shows** $([m,p] \vdash ConstantExpr\ c \mapsto v) = (valid\text{-}value\ v\ (constantAsStamp\ c) \wedge v = c)$
**by** *blast*

**lemma** *unfold-binary*:
**shows** $([m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto val) = (\exists\ x\ y.$
$(([m,p] \vdash xe \mapsto x) \wedge$
$([m,p] \vdash ye \mapsto y) \wedge$
$(val = bin\text{-}eval\ op\ x\ y) \wedge$
$(val \neq UndefVal)$
$))$ (**is** $?L = ?R$)
**proof** (*intro iffI*)
**assume** *3*: $?L$
**show** $?R$ **by** (*rule evaltree.cases*[*OF 3*]; *blast+*)
**next**
**assume** $?R$
**then obtain** *x y* **where** $[m,p] \vdash xe \mapsto x$
**and** $[m,p] \vdash ye \mapsto y$
**and** $val = bin\text{-}eval\ op\ x\ y$
**and** $val \neq UndefVal$
**by** *auto*
**then show** $?L$
**by** (*rule BinaryExpr*)
**qed**

**lemma** *unfold-unary*:
**shows** $([m,p] \vdash UnaryExpr\ op\ xe \mapsto val)$
$= (\exists\ x.$
$(([m,p] \vdash xe \mapsto x) \wedge$
$(val = unary\text{-}eval\ op\ x) \wedge$
$(val \neq UndefVal)$
$))$ (**is** $?L = ?R$)

**by** *auto*

**lemmas** *unfold-evaltree* =
*unfold-binary*
*unfold-unary*

## 1.7 Lemmas about *new_int* and integer eval results.

**lemma** *unary-eval-new-int*:
**assumes** *def*: *unary-eval op x* ≠ *UndefVal*
**shows** ∃ *b v*. *unary-eval op x* = *new-int b v* ∧
*b* = (*if op* ∈ *normal-unary then intval-bits x else ir-resultBits op*)
**proof** (*cases op* ∈ *normal-unary*)
**case** *True*
**then show** *?thesis*
**by** (*metis def empty-iff insert-iff intval-abs.elims intval-bits.simps intval-logic-negation.elims intval-negate.elims intval-not.elims unary-eval.simps*(*1*) *unary-eval.simps*(*2*) *unary-eval.simps*(*3*) *unary-eval.simps*(*4*))
**next**
**case** *False*
**consider** *ib ob* **where** *op* = *UnaryNarrow ib ob* |
*ib ob* **where** *op* = *UnaryZeroExtend ib ob* |
*ib ob* **where** *op* = *UnarySignExtend ib ob*
**by** (*metis False IRUnaryOp.exhaust insert-iff*)
**then show** *?thesis*
**proof** (*cases*)
**case** *1*
**then show** *?thesis*
**by** (*metis False IRUnaryOp.sel*(*4*) *def intval-narrow.elims unary-eval.simps*(*5*))
**next**
**case** *2*
**then show** *?thesis*
**by** (*metis False IRUnaryOp.sel*(*6*) *def intval-zero-extend.elims unary-eval.simps*(*7*))
**next**
**case** *3*
**then show** *?thesis*
**by** (*metis False IRUnaryOp.sel*(*5*) *def intval-sign-extend.elims unary-eval.simps*(*6*))
**qed**
**qed**

**lemma** *new-int-unused-bits-zero*:
**assumes** *IntVal b ival* = *new-int b ival0*
**shows** *take-bit b ival* = *ival*
**using** *assms*(*1*) *new-int-take-bits* **by** *blast*

**lemma** *unary-eval-unused-bits-zero*:
**assumes** *unary-eval op x* = *IntVal b ival*

**shows** *take-bit b ival = ival*
**using** *assms unary-eval-new-int*
**by** (*metis Value.inject*(*1*) *Value.simps*(*5*) *new-int.elims new-int-unused-bits-zero*)

**lemma** *bin-eval-unused-bits-zero*:
**assumes** *bin-eval op x y* = (*IntVal b ival*)
**shows** *take-bit b ival = ival*
**using** *assms bin-eval-new-int*
**by** (*metis Value.distinct*(*1*) *Value.inject*(*1*) *new-int.elims new-int-take-bits*)

**lemma** *eval-unused-bits-zero*:
$[m,p] \vdash xe \mapsto (IntVal\ b\ ix) \Longrightarrow take\text{-}bit\ b\ ix = ix$
**proof** (*induction xe*)
**case** (*UnaryExpr x1 xe*)
**then show** *?case*
**using** *unary-eval-unused-bits-zero* **by** *force*
**next**
**case** (*BinaryExpr x1 xe1 xe2*)
**then show** *?case*
**using** *bin-eval-unused-bits-zero* **by** *force*
**next**
**case** (*ConditionalExpr xe1 xe2 xe3*)
**then show** *?case*
**by** (*metis* (*full-types*) *EvalTreeE*(*3*))
**next**
**case** (*ParameterExpr i s*)
**then have** *valid-value* (*p*!*i*) *s*
**by** *fastforce*
**then show** *?case*
**by** (*metis ParameterExprE Value.distinct*(*7*) *intval-bits.simps intval-word.simps local.ParameterExpr valid-value.elims*(*2*))
**next**
**case** (*LeafExpr x1 x2*)
**then show** *?case*
**by** (*smt* (*z3*) *EvalTreeE*(*6*) *Value.simps*(*11*) *valid-value.elims*(*1*) *valid-value.simps*(*1*))

**next**
**case** (*ConstantExpr x*)
**then show** *?case*
**by** (*metis EvalTreeE*(*1*) *constantAsStamp.simps*(*1*) *valid-value.simps*(*1*))
**next**
**case** (*ConstantVar x*)
**then show** *?case*
**by** *fastforce*
**next**
**case** (*VariableExpr x1 x2*)
**then show** *?case*
**by** *fastforce*
**qed**

**lemma** *unary-normal-bitsize*:
  **assumes** *unary-eval op x = IntVal b ival*
  **assumes** *op ∈ normal-unary*
  **shows** ∃ *ix. x = IntVal b ix*
  **apply** (*cases op*)
        **prefer** *7* **using** *assms* **apply** *blast*
      **prefer** *6* **using** *assms* **apply** *blast*
    **prefer** *5* **using** *assms* **apply** *blast*
  **using** *Value.distinct*(*1*) *Value.sel*(*1*) *assms*(*1*) *new-int.simps unary-eval.simps*
    *intval-abs.elims intval-negate.elims intval-not.elims intval-logic-negation.elims*
    **apply** *metis+*
  **done**

**lemma** *unary-not-normal-bitsize*:
  **assumes** *unary-eval op x = IntVal b ival*
  **assumes** *op ∉ normal-unary*
  **shows** *b = ir-resultBits op* ∧ *0 < b* ∧ *b ≤ 64*
  **apply** (*cases op*)
  **using** *assms* **apply** *blast+*
   **apply** (*metis IRUnaryOp.sel*(*4*) *Value.distinct*(*1*) *Value.sel*(*1*) *assms*(*1*) *intval-narrow.elims intval-narrow-ok new-int.simps unary-eval.simps*(*5*))
   **apply** (*smt* (*verit*) *IRUnaryOp.sel*(*5*) *Value.distinct*(*1*) *Value.sel*(*1*) *assms*(*1*) *intval-sign-extend.elims new-int.simps order-less-le-trans unary-eval.simps*(*6*))
  **apply** (*metis IRUnaryOp.sel*(*6*) *Value.distinct*(*1*) *assms*(*1*) *intval-bits.simps intval-zero-extend.elims linorder-not-less neq0-conv new-int.simps unary-eval.simps*(*7*))
  **done**

**lemma** *unary-eval-bitsize*:
  **assumes** *unary-eval op x = IntVal b ival*
  **assumes** *2*: *x = IntVal bx ix*
  **assumes** *0 < bx* ∧ *bx ≤ 64*
  **shows** *0 < b* ∧ *b ≤ 64*
**proof** (*cases op ∈ normal-unary*)
  **case** *True*
  **then obtain** *tmp* **where** *unary-eval op x = new-int bx tmp*
    **by** (*cases op*; *simp*; *auto simp*: *2*)
  **then show** *?thesis*
    **using** *assms* **by** *simp*
**next**
  **case** *False*
  **then obtain** *tmp* **where** *unary-eval op x = new-int b tmp* ∧ *0 < b* ∧ *b ≤ 64*
    **apply** (*cases op*; *simp*; *auto simp*: *2*)
   **apply** (*metis 2 Value.inject*(*1*) *Value.simps*(*5*) *assms*(*1*) *intval-narrow.simps*(*1*) *intval-narrow-ok new-int.simps unary-eval.simps*(*5*))
   **apply** (*metis 2 Value.distinct*(*1*) *Value.inject*(*1*) *assms*(*1*) *bot-nat-0.not-eq-extremum diff-is-0-eq intval-sign-extend.elims new-int.simps unary-eval.simps*(*6*) *zero-less-diff*)

**by** (*smt* (*verit*, *del-insts*) *2* *Value.simps*(*5*) *assms*(*1*) *intval-bits.simps* *intval-zero-extend.simps*(*1*) *new-int.simps* *order-less-le-trans* *unary-eval.simps*(*7*))
**then show** *?thesis*
**by** *blast*
**qed**

**lemma** *bin-eval-inputs-are-ints*:
**assumes** *bin-eval op x y* = *IntVal b ix*
**obtains** *xb yb xi yi* **where** $x = IntVal\ xb\ xi \wedge y = IntVal\ yb\ yi$
**proof** −
**have** *bin-eval op x y* ≠ *UndefVal*
**by** (*simp add*: *assms*)
**then show** *?thesis*
**using** *assms* **apply** (*cases op*; *cases x*; *cases y*; *simp*)
**using** *that* **by** *blast+*
**qed**

**lemma** *eval-bits-1-64*:
$[m,p] \vdash xe \mapsto (IntVal\ b\ ix) \Longrightarrow 0 < b \wedge b \leq 64$
**proof** (*induction xe arbitrary*: *b ix*)
**case** (*UnaryExpr op x2*)
**then obtain** *xv* **where**
*xv*: $([m,p] \vdash x2 \mapsto xv) \wedge$
*IntVal b ix* = *unary-eval op xv*
**using** *unfold-binary* **by** *auto*
**then have** *b* = (*if op* ∈ *normal-unary then intval-bits xv else ir-resultBits op*)
**using** *unary-eval-new-int*
**by** (*metis Value.disc*(*1*) *Value.discI*(*1*) *Value.sel*(*1*) *new-int.simps*)
**then show** *?case*
**by** (*metis xv UnaryExpr.IH unary-normal-bitsize unary-not-normal-bitsize*)
**next**
**case** (*BinaryExpr op x y*)
**then obtain** *xv yv* **where**
*xy*: $([m,p] \vdash x \mapsto xv) \wedge$
$([m,p] \vdash y \mapsto yv) \wedge$
*IntVal b ix* = *bin-eval op xv yv*
**using** *unfold-binary* **by** *auto*
**then have** *def*: *bin-eval op xv yv* ≠ *UndefVal* **and** *xv*: *xv* ≠ *UndefVal* **and** *yv* ≠ *UndefVal*
**using** *evaltree-not-undef xy* **by** (*force*, *blast*, *blast*)
**then have** *b* = (*if op* ∈ *fixed-32 then 32 else intval-bits xv*)
**by** (*metis xy intval-bits.simps new-int.simps bin-eval-new-int*)
**then show** *?case*
**by** (*metis BinaryExpr.IH*(*1*) *Value.distinct*(*7*) *Value.distinct*(*9*) *xv bin-eval-inputs-are-ints intval-bits.elims le-add-same-cancel1 less-or-eq-imp-le numeral-Bit0 xy zero-less-numeral*)

```
next
  case (ConditionalExpr xe1 xe2 xe3)
  then show ?case
    by (metis (full-types) EvalTreeE(3))
next
  case (ParameterExpr x1 x2)
  then show ?case
   using ParameterExprE intval-bits.simps valid-stamp.simps(1) valid-value.elims(2)
valid-value.simps(17)
    by (metis (no-types, lifting))
next
  case (LeafExpr x1 x2)
  then show ?case
   by (smt (z3) EvalTreeE(6) Value.distinct(7) Value.inject(1) valid-stamp.simps(1)
valid-value.elims(1))
next
  case (ConstantExpr x)
  then show ?case
   by (metis EvalTreeE(1) constantAsStamp.simps(1) valid-stamp.simps(1) valid-value.simps(1))

next
  case (ConstantVar x)
  then show ?case
    by blast
next
  case (VariableExpr x1 x2)
  then show ?case
    by blast
qed


end
```

# 2 Tree to Graph

```
theory TreeToGraph
  imports
    Semantics.IRTreeEval
    Graph.IRGraph
begin
```

## 2.1 Subgraph to Data-flow Tree

```
fun find-node-and-stamp :: IRGraph ⇒ (IRNode × Stamp) ⇒ ID option where
  find-node-and-stamp g (n,s) =
     find (λi. kind g i = n ∧ stamp g i = s) (sorted-list-of-set(ids g))

export-code find-node-and-stamp
```

```
fun is-preevaluated :: IRNode ⇒ bool where
  is-preevaluated (InvokeNode n - - - - -) = True |
  is-preevaluated (InvokeWithExceptionNode n - - - - - -) = True |
  is-preevaluated (NewInstanceNode n - - -) = True |
  is-preevaluated (LoadFieldNode n - - -) = True |
  is-preevaluated (SignedDivNode n - - - - -) = True |
  is-preevaluated (SignedRemNode n - - - - -) = True |
  is-preevaluated (ValuePhiNode n - -) = True |
  is-preevaluated - = False
```

```
inductive
  rep :: IRGraph ⇒ ID ⇒ IRExpr ⇒ bool (- ⊢ - ≃ - 55)
  for g where

  ConstantNode:
  ⟦kind g n = ConstantNode c⟧
    ⟹ g ⊢ n ≃ (ConstantExpr c) |

  ParameterNode:
  ⟦kind g n = ParameterNode i;
    stamp g n = s⟧
    ⟹ g ⊢ n ≃ (ParameterExpr i s) |

  ConditionalNode:
  ⟦kind g n = ConditionalNode c t f;
    g ⊢ c ≃ ce;
    g ⊢ t ≃ te;
    g ⊢ f ≃ fe⟧
    ⟹ g ⊢ n ≃ (ConditionalExpr ce te fe) |


  AbsNode:
  ⟦kind g n = AbsNode x;
    g ⊢ x ≃ xe⟧
    ⟹ g ⊢ n ≃ (UnaryExpr UnaryAbs xe) |

  NotNode:
  ⟦kind g n = NotNode x;
    g ⊢ x ≃ xe⟧
    ⟹ g ⊢ n ≃ (UnaryExpr UnaryNot xe) |

  NegateNode:
  ⟦kind g n = NegateNode x;
    g ⊢ x ≃ xe⟧
    ⟹ g ⊢ n ≃ (UnaryExpr UnaryNeg xe) |

  LogicNegationNode:
```

$[\![kind\ g\ n = LogicNegationNode\ x;$
$\ \ g \vdash x \simeq xe]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryLogicNegation\ xe)\ |$

$AddNode$:
$[\![kind\ g\ n = AddNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinAdd\ xe\ ye)\ |$

$MulNode$:
$[\![kind\ g\ n = MulNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinMul\ xe\ ye)\ |$

$SubNode$:
$[\![kind\ g\ n = SubNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinSub\ xe\ ye)\ |$

$AndNode$:
$[\![kind\ g\ n = AndNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinAnd\ xe\ ye)\ |$

$OrNode$:
$[\![kind\ g\ n = OrNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinOr\ xe\ ye)\ |$

$XorNode$:
$[\![kind\ g\ n = XorNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinXor\ xe\ ye)\ |$

$ShortCircuitOrNode$:
$[\![kind\ g\ n = ShortCircuitOrNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye]\!]$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinShortCircuitOr\ xe\ ye)\ |$

$LeftShiftNode$:
$[\![kind\ g\ n = LeftShiftNode\ x\ y;$

$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinLeftShift\ xe\ ye) \mid$

*RightShiftNode*:
$[\![kind\ g\ n = RightShiftNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinRightShift\ xe\ ye) \mid$

*UnsignedRightShiftNode*:
$[\![kind\ g\ n = UnsignedRightShiftNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinURightShift\ xe\ ye) \mid$

*IntegerBelowNode*:
$[\![kind\ g\ n = IntegerBelowNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerBelow\ xe\ ye) \mid$

*IntegerEqualsNode*:
$[\![kind\ g\ n = IntegerEqualsNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerEquals\ xe\ ye) \mid$

*IntegerLessThanNode*:
$[\![kind\ g\ n = IntegerLessThanNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerLessThan\ xe\ ye) \mid$

*NarrowNode*:
$[\![kind\ g\ n = NarrowNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)\ xe) \mid$

*SignExtendNode*:
$[\![kind\ g\ n = SignExtendNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnarySignExtend\ inputBits\ resultBits)\ xe) \mid$

*ZeroExtendNode*:
$[\![kind\ g\ n = ZeroExtendNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnaryZeroExtend\ inputBits\ resultBits)\ xe) \mid$

*LeafNode*:
$[\![$*is-preevaluated* (*kind g n*);
*stamp g n* = *s*$]\!]$
$\Longrightarrow g \vdash n \simeq$ (*LeafExpr n s*) |

*RefNode*:
$[\![$*kind g n* = *RefNode n'*;
$g \vdash n' \simeq e]\!]$
$\Longrightarrow g \vdash n \simeq e$

**code-pred** (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool as exprE*) *rep* **.**

**inductive**
*replist* :: *IRGraph* $\Rightarrow$ *ID list* $\Rightarrow$ *IRExpr list* $\Rightarrow$ *bool* (- $\vdash$ - $\simeq_L$ - *55*)
**for** *g* **where**

*RepNil*:
$g \vdash [] \simeq_L []$ |

*RepCons*:
$[\![g \vdash x \simeq xe;$
$g \vdash xs \simeq_L xse]\!]$
$\Longrightarrow g \vdash x\#xs \simeq_L xe\#xse$

**code-pred** (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool as exprListE*) *replist* **.**

**definition** *wf-term-graph* :: *MapState* $\Rightarrow$ *Params* $\Rightarrow$ *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *bool* **where**
*wf-term-graph m p g n* = ($\exists$ *e*. ($g \vdash n \simeq e$) $\wedge$ ($\exists$ *v*. ([*m*, *p*] $\vdash e \mapsto v$)))

**values** $\{t.$ *eg2-sq* $\vdash 4 \simeq t\}$

## 2.2 Data-flow Tree to Subgraph

**fun** *unary-node* :: *IRUnaryOp* $\Rightarrow$ *ID* $\Rightarrow$ *IRNode* **where**
*unary-node UnaryAbs v* = *AbsNode v* |
*unary-node UnaryNot v* = *NotNode v* |
*unary-node UnaryNeg v* = *NegateNode v* |
*unary-node UnaryLogicNegation v* = *LogicNegationNode v* |
*unary-node* (*UnaryNarrow ib rb*) *v* = *NarrowNode ib rb v* |
*unary-node* (*UnarySignExtend ib rb*) *v* = *SignExtendNode ib rb v* |
*unary-node* (*UnaryZeroExtend ib rb*) *v* = *ZeroExtendNode ib rb v*

**fun** *bin-node* :: *IRBinaryOp* ⇒ *ID* ⇒ *ID* ⇒ *IRNode* **where**
*bin-node BinAdd x y = AddNode x y* |
*bin-node BinMul x y = MulNode x y* |
*bin-node BinSub x y = SubNode x y* |
*bin-node BinAnd x y = AndNode x y* |
*bin-node BinOr x y = OrNode x y* |
*bin-node BinXor x y = XorNode x y* |
*bin-node BinShortCircuitOr x y = ShortCircuitOrNode x y* |
*bin-node BinLeftShift x y = LeftShiftNode x y* |
*bin-node BinRightShift x y = RightShiftNode x y* |
*bin-node BinURightShift x y = UnsignedRightShiftNode x y* |
*bin-node BinIntegerEquals x y = IntegerEqualsNode x y* |
*bin-node BinIntegerLessThan x y = IntegerLessThanNode x y* |
*bin-node BinIntegerBelow x y = IntegerBelowNode x y*

**inductive** *fresh-id* :: *IRGraph* ⇒ *ID* ⇒ *bool* **where**
$n \notin$ *ids g* ⟹ *fresh-id g n*

**code-pred** *fresh-id* **.**

**fun** *get-fresh-id* :: *IRGraph* ⇒ *ID* **where**

*get-fresh-id g = last(sorted-list-of-set(ids g)) + 1*

**export-code** *get-fresh-id*

**value** *get-fresh-id eg2-sq*
**value** *get-fresh-id* (*add-node 6* (*ParameterNode 2*, *default-stamp*) *eg2-sq*)

**inductive**
*unrep* :: *IRGraph* ⇒ *IRExpr* ⇒ (*IRGraph* × *ID*) ⇒ *bool* (- ⊕ - ⇝ - *55*)
**where**

*ConstantNodeSame*:
⟦*find-node-and-stamp g* (*ConstantNode c*, *constantAsStamp c*) = *Some n*⟧
⟹ *g* ⊕ (*ConstantExpr c*) ⇝ (*g*, *n*) |

*ConstantNodeNew*:
⟦*find-node-and-stamp g* (*ConstantNode c*, *constantAsStamp c*) = *None*;
*n* = *get-fresh-id g*;
*g′* = *add-node n* (*ConstantNode c*, *constantAsStamp c*) *g* ⟧
⟹ *g* ⊕ (*ConstantExpr c*) ⇝ (*g′*, *n*) |

*ParameterNodeSame*:
⟦*find-node-and-stamp g* (*ParameterNode i*, *s*) = *Some n*⟧

$\Longrightarrow g \oplus (\mathit{ParameterExpr}\ i\ s) \rightsquigarrow (g,\ n)$ |

*ParameterNodeNew*:

$[\![\mathit{find\text{-}node\text{-}and\text{-}stamp}\ g\ (\mathit{ParameterNode}\ i,\ s) = \mathit{None};$
$n = \mathit{get\text{-}fresh\text{-}id}\ g;$
$g' = \mathit{add\text{-}node}\ n\ (\mathit{ParameterNode}\ i,\ s)\ g]\!]$
$\Longrightarrow g \oplus (\mathit{ParameterExpr}\ i\ s) \rightsquigarrow (g',\ n)$ |

*ConditionalNodeSame*:

$[\![g \oplus ce \rightsquigarrow (g2,\ c);$
$g2 \oplus te \rightsquigarrow (g3,\ t);$
$g3 \oplus fe \rightsquigarrow (g4,\ f);$
$s' = \mathit{meet}\ (\mathit{stamp}\ g4\ t)\ (\mathit{stamp}\ g4\ f);$
$\mathit{find\text{-}node\text{-}and\text{-}stamp}\ g4\ (\mathit{ConditionalNode}\ c\ t\ f,\ s') = \mathit{Some}\ n]\!]$
$\Longrightarrow g \oplus (\mathit{ConditionalExpr}\ ce\ te\ fe) \rightsquigarrow (g4,\ n)$ |

*ConditionalNodeNew*:

$[\![g \oplus ce \rightsquigarrow (g2,\ c);$
$g2 \oplus te \rightsquigarrow (g3,\ t);$
$g3 \oplus fe \rightsquigarrow (g4,\ f);$
$s' = \mathit{meet}\ (\mathit{stamp}\ g4\ t)\ (\mathit{stamp}\ g4\ f);$
$\mathit{find\text{-}node\text{-}and\text{-}stamp}\ g4\ (\mathit{ConditionalNode}\ c\ t\ f,\ s') = \mathit{None};$
$n = \mathit{get\text{-}fresh\text{-}id}\ g4;$
$g' = \mathit{add\text{-}node}\ n\ (\mathit{ConditionalNode}\ c\ t\ f,\ s')\ g4]\!]$
$\Longrightarrow g \oplus (\mathit{ConditionalExpr}\ ce\ te\ fe) \rightsquigarrow (g',\ n)$ |

*UnaryNodeSame*:

$[\![g \oplus xe \rightsquigarrow (g2,\ x);$
$s' = \mathit{stamp\text{-}unary}\ op\ (\mathit{stamp}\ g2\ x);$
$\mathit{find\text{-}node\text{-}and\text{-}stamp}\ g2\ (\mathit{unary\text{-}node}\ op\ x,\ s') = \mathit{Some}\ n]\!]$
$\Longrightarrow g \oplus (\mathit{UnaryExpr}\ op\ xe) \rightsquigarrow (g2,\ n)$ |

*UnaryNodeNew*:

$[\![g \oplus xe \rightsquigarrow (g2,\ x);$
$s' = \mathit{stamp\text{-}unary}\ op\ (\mathit{stamp}\ g2\ x);$
$\mathit{find\text{-}node\text{-}and\text{-}stamp}\ g2\ (\mathit{unary\text{-}node}\ op\ x,\ s') = \mathit{None};$
$n = \mathit{get\text{-}fresh\text{-}id}\ g2;$
$g' = \mathit{add\text{-}node}\ n\ (\mathit{unary\text{-}node}\ op\ x,\ s')\ g2]\!]$
$\Longrightarrow g \oplus (\mathit{UnaryExpr}\ op\ xe) \rightsquigarrow (g',\ n)$ |

*BinaryNodeSame*:

$[\![g \oplus xe \rightsquigarrow (g2,\ x);$
$g2 \oplus ye \rightsquigarrow (g3,\ y);$
$s' = \mathit{stamp\text{-}binary}\ op\ (\mathit{stamp}\ g3\ x)\ (\mathit{stamp}\ g3\ y);$
$\mathit{find\text{-}node\text{-}and\text{-}stamp}\ g3\ (\mathit{bin\text{-}node}\ op\ x\ y,\ s') = \mathit{Some}\ n]\!]$
$\Longrightarrow g \oplus (\mathit{BinaryExpr}\ op\ xe\ ye) \rightsquigarrow (g3,\ n)$ |

*BinaryNodeNew*:

$[\![g \oplus xe \rightsquigarrow (g2,\ x);$

$g2 \oplus ye \rightsquigarrow (g3,\ y)$;
$s' = stamp\text{-}binary\ op\ (stamp\ g3\ x)\ (stamp\ g3\ y)$;
$find\text{-}node\text{-}and\text{-}stamp\ g3\ (bin\text{-}node\ op\ x\ y,\ s') = None$;
$n = get\text{-}fresh\text{-}id\ g3$;
$g' = add\text{-}node\ n\ (bin\text{-}node\ op\ x\ y,\ s')\ g3]\!]$
$\Longrightarrow g \oplus (BinaryExpr\ op\ xe\ ye) \rightsquigarrow (g',\ n)$ |

*AllLeafNodes*:
$[\![stamp\ g\ n = s$;
$is\text{-}preevaluated\ (kind\ g\ n)]\!]$
$\Longrightarrow g \oplus (LeafExpr\ n\ s) \rightsquigarrow (g,\ n)$

**code-pred** (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$ *as unrepE*)
*unrep* **.**

*unrepRules*

$$\frac{\textit{find-node-and-stamp } g \ (\textit{ConstantNode } c,\ \textit{constantAsStamp } c) = \textit{Some } n}{g \oplus \textit{ConstantExpr } c \rightsquigarrow (g,\ n)}$$

$$\frac{\begin{array}{c}\textit{find-node-and-stamp } g \ (\textit{ConstantNode } c,\ \textit{constantAsStamp } c) = \textit{None} \\ n = \textit{get-fresh-id } g \\ g' = \textit{add-node } n \ (\textit{ConstantNode } c,\ \textit{constantAsStamp } c) \ g\end{array}}{g \oplus \textit{ConstantExpr } c \rightsquigarrow (g',\ n)}$$

$$\frac{\textit{find-node-and-stamp } g \ (\textit{ParameterNode } i,\ s) = \textit{Some } n}{g \oplus \textit{ParameterExpr } i \ s \rightsquigarrow (g,\ n)}$$

$$\frac{\begin{array}{c}\textit{find-node-and-stamp } g \ (\textit{ParameterNode } i,\ s) = \textit{None} \\ n = \textit{get-fresh-id } g \qquad g' = \textit{add-node } n \ (\textit{ParameterNode } i,\ s) \ g\end{array}}{g \oplus \textit{ParameterExpr } i \ s \rightsquigarrow (g',\ n)}$$

$$\frac{\begin{array}{c}g \oplus ce \rightsquigarrow (g2,\ c) \qquad g2 \oplus te \rightsquigarrow (g3,\ t) \\ g3 \oplus fe \rightsquigarrow (g4,\ f) \qquad s' = \textit{meet } (\textit{stamp } g4 \ t) \ (\textit{stamp } g4 \ f) \\ \textit{find-node-and-stamp } g4 \ (\textit{ConditionalNode } c \ t \ f,\ s') = \textit{Some } n\end{array}}{g \oplus \textit{ConditionalExpr } ce \ te \ fe \rightsquigarrow (g4,\ n)}$$

$$\frac{\begin{array}{c}g \oplus ce \rightsquigarrow (g2,\ c) \qquad g2 \oplus te \rightsquigarrow (g3,\ t) \\ g3 \oplus fe \rightsquigarrow (g4,\ f) \qquad s' = \textit{meet } (\textit{stamp } g4 \ t) \ (\textit{stamp } g4 \ f) \\ \textit{find-node-and-stamp } g4 \ (\textit{ConditionalNode } c \ t \ f,\ s') = \textit{None} \\ n = \textit{get-fresh-id } g4 \qquad g' = \textit{add-node } n \ (\textit{ConditionalNode } c \ t \ f,\ s') \ g4\end{array}}{g \oplus \textit{ConditionalExpr } ce \ te \ fe \rightsquigarrow (g',\ n)}$$

$$\frac{\begin{array}{c}g \oplus xe \rightsquigarrow (g2,\ x) \\ g2 \oplus ye \rightsquigarrow (g3,\ y) \qquad s' = \textit{stamp-binary } op \ (\textit{stamp } g3 \ x) \ (\textit{stamp } g3 \ y) \\ \textit{find-node-and-stamp } g3 \ (\textit{bin-node } op \ x \ y,\ s') = \textit{Some } n\end{array}}{g \oplus \textit{BinaryExpr } op \ xe \ ye \rightsquigarrow (g3,\ n)}$$

$$\frac{\begin{array}{c}g \oplus xe \rightsquigarrow (g2,\ x) \\ g2 \oplus ye \rightsquigarrow (g3,\ y) \qquad s' = \textit{stamp-binary } op \ (\textit{stamp } g3 \ x) \ (\textit{stamp } g3 \ y) \\ \textit{find-node-and-stamp } g3 \ (\textit{bin-node } op \ x \ y,\ s') = \textit{None} \\ n = \textit{get-fresh-id } g3 \qquad g' = \textit{add-node } n \ (\textit{bin-node } op \ x \ y,\ s') \ g3\end{array}}{g \oplus \textit{BinaryExpr } op \ xe \ ye \rightsquigarrow (g',\ n)}$$

$$\frac{\begin{array}{c}g \oplus xe \rightsquigarrow (g2,\ x) \qquad s' = \textit{stamp-unary } op \ (\textit{stamp } g2 \ x) \\ \textit{find-node-and-stamp } g2 \ (\textit{unary-node } op \ x,\ s') = \textit{Some } n\end{array}}{g \oplus \textit{UnaryExpr } op \ xe \rightsquigarrow (g2,\ n)}$$

$$\frac{\begin{array}{c}g \oplus xe \rightsquigarrow (g2,\ x) \qquad s' = \textit{stamp-unary } op \ (\textit{stamp } g2 \ x) \\ \textit{find-node-and-stamp } g2 \ (\textit{unary-node } op \ x,\ s') = \textit{None} \\ n = \textit{get-fresh-id } g2 \qquad g' = \textit{add-node } n \ (\textit{unary-node } op \ x,\ s') \ g2\end{array}}{g \oplus \textit{UnaryExpr } op \ xe \rightsquigarrow (g',\ n)}$$

$$\frac{\textit{stamp } g \ n = s \qquad \textit{is-preevaluated } (\textit{kind } g \ n)}{g \oplus \textit{LeafExpr } n \ s \rightsquigarrow (g,\ n)}$$

*values* $\{(n,\ g)\ .\ (eg2\text{-}sq \oplus sq\text{-}param0 \rightsquigarrow (g,\ n))\}$

## 2.3 Lift Data-flow Tree Semantics

**definition** *encodeeval* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID* ⇒ *Value* ⇒ *bool*
  ([-,-,-] ⊢ - ↦ - *50*)
  **where**
  $encodeeval\ g\ m\ p\ n\ v = (\exists\ e.\ (g \vdash n \simeq e) \wedge ([m,p] \vdash e \mapsto v))$

## 2.4 Graph Refinement

**definition** *graph-represents-expression* :: *IRGraph* ⇒ *ID* ⇒ *IRExpr* ⇒ *bool*
  (- ⊢ - ⊴ - *50*)
  **where**
  $(g \vdash n \trianglelefteq e) = (\exists\, e'\ .\ (g \vdash n \simeq e') \wedge (e' \leq e))$

**definition** *graph-refinement* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
  $graph\text{-}refinement\ g_1\ g_2 =$
    $((ids\ g_1 \subseteq ids\ g_2) \wedge$
    $(\forall\ n\ .\ n \in ids\ g_1 \longrightarrow (\forall\, e.\ (g_1 \vdash n \simeq e) \longrightarrow (g_2 \vdash n \trianglelefteq e))))$

**lemma** *graph-refinement*:
  $graph\text{-}refinement\ g1\ g2 \Longrightarrow (\forall\, n\ m\ p\ v.\ n \in ids\ g1 \longrightarrow ([g1,\ m,\ p] \vdash n \mapsto v) \longrightarrow ([g2,\ m,\ p] \vdash n \mapsto v))$
   **by** (*meson encodeeval-def graph-refinement-def graph-represents-expression-def le-expr-def*)

## 2.5 Maximal Sharing

**definition** *maximal-sharing*:
  $maximal\text{-}sharing\ g = (\forall\ n_1\ n_2\ .\ n_1 \in true\text{-}ids\ g \wedge n_2 \in true\text{-}ids\ g \longrightarrow$
   $(\forall\ e.\ (g \vdash n_1 \simeq e) \wedge (g \vdash n_2 \simeq e) \wedge (stamp\ g\ n_1 = stamp\ g\ n_2) \longrightarrow n_1 = n_2))$

**end**