

# Veriopt Theories

September 6, 2022

## Contents

<b>1</b>	<b>Data-flow Semantics</b>	<b>1</b>
1.1	Data-flow Tree Representation . . . . .	1
1.2	Functions for re-calculating stamps . . . . .	3
1.3	Data-flow Tree Evaluation . . . . .	4
1.4	Data-flow Tree Refinement . . . . .	6
1.5	Stamp Masks . . . . .	7
1.6	Data-flow Tree Theorems . . . . .	8
1.6.1	Deterministic Data-flow Evaluation . . . . .	8
1.6.2	Typing Properties for Integer Evaluation Functions . .	9
1.6.3	Evaluation Results are Valid . . . . .	11
1.6.4	Example Data-flow Optimisations . . . . .	12
1.6.5	Monotonicity of Expression Refinement . . . . .	12
1.7	Unfolding rules for evaltree quadruples down to bin-eval level	14
1.8	Lemmas about <i>new_int</i> and integer eval results. . . . .	15
<b>2</b>	<b>Tree to Graph</b>	<b>19</b>
2.1	Subgraph to Data-flow Tree . . . . .	19
2.2	Data-flow Tree to Subgraph . . . . .	23
2.3	Lift Data-flow Tree Semantics . . . . .	28
2.4	Graph Refinement . . . . .	28
2.5	Maximal Sharing . . . . .	28

## 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 calculated 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*  $\Rightarrow$  *Value*  
**type-synonym** *Params* = *Value list*

**definition** *new-map-state* :: *MapState* **where**  
*new-map-state* = ( $\lambda x.$  *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. Many operators have the same output bits as their inputs. However, the unary integer operators that are not *normal\_unary* are narrowing or widening operators, so the result bits is specified by the operator. The binary integer operators are divided into three groups: (1) *binary\_fixed\_32* operators always output 32 bits, (2) *binary\_shift\_ops* operators output size is determined by their left argument, and (3) other operators output the same number of bits as both their inputs.

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

**abbreviation** *binary-shift-ops* :: *IRBinaryOp* *set* **where**  
*binary-shift-ops* ≡ { *BinLeftShift*, *BinRightShift*, *BinURightShift* }

**abbreviation** *normal-unary* :: *IRUnaryOp* *set* **where**

```

normal-unary  $\equiv \{ \text{UnaryAbs}, \text{UnaryNeg}, \text{UnaryNot}, \text{UnaryLogicNegation} \}$ 

fun stamp-unary :: IRUnaryOp  $\Rightarrow$  Stamp  $\Rightarrow$  Stamp where

  stamp-unary op (IntegerStamp b lo hi) =
    unrestricted-stamp (IntegerStamp (if op  $\in$  normal-unary then b else (ir-resultBits
    op)) lo hi) |

  stamp-unary op - = IllegalStamp

fun stamp-binary :: IRBinaryOp  $\Rightarrow$  Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  Stamp where
  stamp-binary op (IntegerStamp b1 lo1 hi1) (IntegerStamp b2 lo2 hi2) =
    (if op  $\in$  binary-shift-ops then unrestricted-stamp (IntegerStamp b1 lo1 hi1)
    else if b1  $\neq$  b2 then IllegalStamp else
    (if op  $\in$  binary-fixed-32-ops
    then unrestricted-stamp (IntegerStamp 32 lo1 hi1)
    else unrestricted-stamp (IntegerStamp b1 lo1 hi1))) |

  stamp-binary op - - = IllegalStamp

fun stamp-expr :: IRExpr  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  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 out-
  Bits v |
  unary-eval (UnaryZeroExtend inBits outBits) v = intval-zero-extend inBits out-
  Bits v

fun bin-eval :: IRBinaryOp  $\Rightarrow$  Value  $\Rightarrow$  Value  $\Rightarrow$  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 |

```

$\text{bin-eval BinAnd } v1 \ v2 = \text{intval-and } v1 \ v2 \mid$   
 $\text{bin-eval BinOr } v1 \ v2 = \text{intval-or } v1 \ v2 \mid$   
 $\text{bin-eval BinXor } v1 \ v2 = \text{intval-xor } v1 \ v2 \mid$   
 $\text{bin-eval BinShortCircuitOr } v1 \ v2 = \text{intval-short-circuit-or } v1 \ v2 \mid$   
 $\text{bin-eval BinLeftShift } v1 \ v2 = \text{intval-left-shift } v1 \ v2 \mid$   
 $\text{bin-eval BinRightShift } v1 \ v2 = \text{intval-right-shift } v1 \ v2 \mid$   
 $\text{bin-eval BinURightShift } v1 \ v2 = \text{intval-uright-shift } v1 \ v2 \mid$   
 $\text{bin-eval BinIntegerEquals } v1 \ v2 = \text{intval-equals } v1 \ v2 \mid$   
 $\text{bin-eval BinIntegerLessThan } v1 \ v2 = \text{intval-less-than } v1 \ v2 \mid$   
 $\text{bin-eval BinIntegerBelow } v1 \ v2 = \text{intval-below } v1 \ v2$

**lemmas** *eval-thms* =

$\text{intval-abs.simps}$   $\text{intval-negate.simps}$   $\text{intval-not.simps}$   
 $\text{intval-logic-negation.simps}$   $\text{intval-narrow.simps}$   
 $\text{intval-sign-extend.simps}$   $\text{intval-zero-extend.simps}$   
 $\text{intval-add.simps}$   $\text{intval-mul.simps}$   $\text{intval-sub.simps}$   
 $\text{intval-and.simps}$   $\text{intval-or.simps}$   $\text{intval-xor.simps}$   
 $\text{intval-left-shift.simps}$   $\text{intval-right-shift.simps}$   
 $\text{intval-uright-shift.simps}$   $\text{intval-equals.simps}$   
 $\text{intval-less-than.simps}$   $\text{intval-below.simps}$

**inductive** *not-undef-or-fail* :: *Value*  $\Rightarrow$  *Value*  $\Rightarrow$  *bool* **where**

$\llbracket \text{value} \neq \text{UndefVal} \rrbracket \Longrightarrow \text{not-undef-or-fail value value}$

**notation** (*latex output*)

*not-undef-or-fail* (- = -)

**inductive**

*evaltree* :: *MapState*  $\Rightarrow$  *Params*  $\Rightarrow$  *IRExpr*  $\Rightarrow$  *Value*  $\Rightarrow$  *bool* ( $[-, -] \vdash - \mapsto -$  55)

**for** *m p* **where**

*ConstantExpr*:

$\llbracket \text{valid-value } c \ (\text{constantAsStamp } c) \rrbracket$   
 $\Longrightarrow [m, p] \vdash (\text{ConstantExpr } c) \mapsto c \mid$

*ParameterExpr*:

$\llbracket i < \text{length } p; \text{valid-value } (p!i) \ s \rrbracket$   
 $\Longrightarrow [m, p] \vdash (\text{ParameterExpr } i \ s) \mapsto p!i \mid$

*ConditionalExpr*:

$\llbracket [m, p] \vdash ce \mapsto \text{cond};$   
 $\text{branch} = (\text{if val-to-bool cond then te else fe});$   
 $[m, p] \vdash \text{branch} \mapsto v;$   
 $v \neq \text{UndefVal} \rrbracket$   
 $\Longrightarrow [m, p] \vdash (\text{ConditionalExpr } ce \ te \ fe) \mapsto v \mid$

*UnaryExpr*:

$\llbracket m, p \rrbracket \vdash xe \mapsto v;$   
 $result = (unary\text{-}eval\ op\ v);$   
 $result \neq UndefinedVal$   
 $\implies [m, p] \vdash (UnaryExpr\ op\ xe) \mapsto result \mid$

*BinaryExpr:*  
 $\llbracket m, p \rrbracket \vdash xe \mapsto x;$   
 $[m, p] \vdash ye \mapsto y;$   
 $result = (bin\text{-}eval\ op\ x\ y);$   
 $result \neq UndefinedVal$   
 $\implies [m, p] \vdash (BinaryExpr\ op\ xe\ ye) \mapsto result \mid$

*LeafExpr:*  
 $\llbracket val = m\ n;$   
 $valid\text{-}value\ val\ s \rrbracket$   
 $\implies [m, p] \vdash LeafExpr\ n\ s \mapsto val$

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

**inductive**

$evaltrees :: MapState \Rightarrow Params \Rightarrow IRExpr\ list \Rightarrow Value\ list \Rightarrow bool\ ([-,] \vdash - \mapsto_L$   
 $- \ 55)$

**for**  $m\ p$  **where**

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

*EvalCons:*  
 $\llbracket [m, p] \vdash x \mapsto xval;$   
 $[m, p] \vdash yy \mapsto_L yyval \rrbracket$   
 $\implies [m, p] \vdash (x \# yy) \mapsto_L (xval \# yyval)$

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

**definition**  $sq\text{-}param0 :: IRExpr$  **where**

$sq\text{-}param0 = BinaryExpr\ BinMul$   
 $(ParameterExpr\ 0\ (IntegerStamp\ 32\ (-\ 2147483648)\ 2147483647))$   
 $(ParameterExpr\ 0\ (IntegerStamp\ 32\ (-\ 2147483648)\ 2147483647))$

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

**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*]:  
 $(e2 \leq e1) \longleftrightarrow (\forall m p v. (([m,p] \vdash e1 \mapsto v) \longrightarrow ([m,p] \vdash e2 \mapsto v)))$

**definition**

*lt-expr-def* [*simp*]:  
 $(e1 < e2) \longleftrightarrow (e1 \leq e2 \wedge \neg (e1 \doteq e2))$

**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**  $e1 \sqsupseteq e2 \equiv (e2 \leq e1)$

## 1.5 Stamp Masks

A stamp can contain additional range information in the form of masks. A stamp has an up mask and a down mask, corresponding to the bits that may be set and the bits that must be set.

Examples: A stamp where no range information is known will have; an up mask of -1 as all bits may be set, and a down mask of 0 as no bits must be set.

A stamp known to be one should have; an up mask of 1 as only the first bit may be set, no others, and a down mask of 1 as the first bit must be set and no others.

We currently don't carry mask information in stamps, and instead assume correct masks to prove optimizations.

```

locale stamp-mask =
  fixes up :: IRExp ⇒ int64 (↑)
  fixes down :: IRExp ⇒ int64 (↓)
  assumes up-spec: [m, p] ⊢ e ↦ IntVal b v ⇒ (and v (not ((ucast (↑e))))) = 0
  and down-spec: [m, p] ⊢ e ↦ IntVal b v ⇒ (and (not v) (ucast (↓e))) = 0
begin

```

```

lemma may-implies-either:
  [m, p] ⊢ e ↦ IntVal b v ⇒ bit (↑e) n ⇒ bit v n = False ∨ bit v n = True
by simp

```

```

lemma not-may-implies-false:
  [m, p] ⊢ e ↦ IntVal b v ⇒ ¬(bit (↑e) n) ⇒ bit v n = False
using up-spec
using bit-and-iff bit-eq-iff bit-not-iff bit-unsigned-iff down-spec
by (smt (verit, best) bit.double-compl)

```

```

lemma must-implies-true:
  [m, p] ⊢ e ↦ IntVal b v ⇒ bit (↓e) n ⇒ bit v n = True
using down-spec
by (metis bit.compl-one bit-and-iff bit-minus-1-iff bit-not-iff impossible-bit ucast-id)

```

```

lemma not-must-implies-either:
  [m, p] ⊢ e ↦ IntVal b v ⇒ ¬(bit (↓e) n) ⇒ bit v n = False ∨ bit v n = True
by simp

```

```

lemma must-implies-may:
  [m, p] ⊢ e ↦ IntVal b v ⇒ n < 32 ⇒ bit (↓e) n ⇒ bit (↑e) n
by (meson must-implies-true not-may-implies-false)

```

```

lemma up-mask-and-zero-implies-zero:
  assumes and (↑x) (↑y) = 0
  assumes [m, p] ⊢ x ↦ IntVal b xv
  assumes [m, p] ⊢ y ↦ IntVal b yv
  shows and xv yv = 0
  using assms
  by (smt (z3) and.commute and.right-neutral and-zero-eq bit.compl-zero bit.conj-cancel-right
    bit.conj-disj-distrib(1) ucast-id up-spec word-bw-assocs(1) word-not-dist(2))

```

```

lemma not-down-up-mask-and-zero-implies-zero:
  assumes and (not (↓x)) (↑y) = 0
  assumes [m, p] ⊢ x ↦ IntVal b xv

```



```

assumes  $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$ 
shows  $\text{and } xv \ yv = yv$ 
using assms
by (smt (z3) and-zero-eq bit.conj-cancel-left bit.conj-disj-distrib(1) bit.conj-disj-distrib(2)
bit.de-Morgan-disj down-spec or-eq-not-not-and ucast-id up-spec word-ao-absorbs(2)
word-ao-absorbs(8) word-bw-lcs(1) word-not-dist(2))

end

end

```

## 1.6 Data-flow Tree Theorems

```

theory IRTreeEvalThms
imports
  Graph.ValueThms
  IRTreeEval
begin

```

### 1.6.1 Deterministic Data-flow Evaluation

```

lemma evalDet:
 $[m, p] \vdash e \mapsto v_1 \implies$ 
 $[m, p] \vdash e \mapsto v_2 \implies$ 
 $v_1 = v_2$ 
apply (induction arbitrary: v2 rule: evaltree.induct)
by (elim EvalTreeE; auto)+

```

```

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

```

### 1.6.2 Typing Properties for Integer Evaluation Functions

We use three simple typing properties on integer values: *isIntVal32*, *isIntVal64* and the more general *isIntVal*.

```

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

```

```

lemma unary-eval-not-obj-str:
shows  $\text{unary-eval } op \ x \neq \text{ObjStr } v$ 
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-int 32 1)
  unfolding new-int.simps
  by auto

lemma bin-eval-new-int:
  assumes def: bin-eval op x y  $\neq$  UndefVal
  shows  $\exists b\ v. (bin-eval\ op\ x\ y) = new-int\ b\ v \wedge$ 
     $b = (if\ op \in binary-fixed-32-ops\ then\ 32\ else\ intval-bits\ x)$ 
  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-bounds b)  $\leq$  int-signed-value b ival  $\wedge$  int-signed-value b ival
 $\leq$  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 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)  $\leq$  int-signed-value b ival  $\wedge$  int-signed-value b ival
 $\leq$  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

```

### 1.6.3 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) ⇒
    (∃ v. val = IntVal b v)
  using valid-value.elims(2) by fastforce

```

```

lemmas valid-value-elim =
  valid-VoidStamp
  valid-ObjStamp
  valid-int

```

```

lemma evaltree-not-undef:
  fixes m p e v
  shows ([m,p] ⊢ e ↦ v) ⇒ v ≠ UndefVal
  apply (induction rule: evaltree.induct)
  using valid-not-undef by auto

```

```

lemma leafint:
  assumes ev: [m,p] ⊢ LeafExpr i (IntegerStamp b lo hi) ↦ val
  shows ∃ 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  $\exists v. (val = \text{IntVal } b \ v \wedge$ 
     $lo \leq \text{int-signed-value } b \ v \wedge$ 
     $\text{int-signed-value } b \ v \leq hi)$ 
  using assms valid-int
  by (metis valid-value.simps(1))

```

#### 1.6.4 Example Data-flow Optimisations

#### 1.6.5 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(UnaryExpr op)*, but it is not obvious how to do this for both arguments of the binary expressions.

```

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

```

```

lemma mono-binary:
  assumes  $x \geq x'$ 
  assumes  $y \geq y'$ 
  shows  $(\text{BinaryExpr op } x \ y) \geq (\text{BinaryExpr op } x' \ y')$ 
  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*:  
 $compatible\ x\ y \wedge compatible\ y\ z \implies compatible\ x\ z$   
**by** (*smt* (*z3*) *compatible.elims*(2) *compatible.simps*(1))

**lemma** *compatible-reft*:  
 $compatible\ x\ y \implies compatible\ y\ x$   
**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**  $b: branch = (if\ val\text{-}to\text{-}bool\ cond\ then\ te\ else\ fe)$   
**define**  $branch'$  **where**  $b': branch' = (if\ val\text{-}to\text{-}bool\ cond\ then\ te'\ else\ fe')$   
**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**

## 1.7 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<sub>e</sub>val* / *unary<sub>e</sub>val* level, simply by saying *unfoldingunfold<sub>e</sub>valtree*.

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

```

```

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

```

```

lemmas unfold-evaltree =
  unfold-binary
  unfold-unary

```

## 1.8 Lemmas about *new\_\_int* and integer eval results.

```

lemma unary-eval-new-int:
assumes def:  $\text{unary-eval op } x \neq \text{UndefVal}$ 
shows  $\exists \ b \ v. \text{unary-eval op } x = \text{new-int } b \ v \wedge$ 
   $b = (\text{if } op \in \text{normal-unary then intval-bits } x \text{ else ir-resultBits } op)$ 
proof (cases  $op \in \text{normal-unary}$ )
  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:
   $[m,p] \vdash xe \mapsto (IntVal\ b\ ix) \implies take-bit\ b\ ix = ix$ 
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) int-

```

```

val-narrow.elims intval-narrow-ok new-int.simps unary-eval.simps(5))
  apply (smt (verit) IRUnaryOp.sel(5) Value.distinct(1) Value.sel(1) asms(1)
intval-sign-extend.elims new-int.simps order-less-le-trans unary-eval.simps(6))
  apply (metis IRUnaryOp.sel(6) Value.distinct(1) asms(1) intval-bits.simps int-
val-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 asms 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) asms(1) intval-narrow.simps(1)
intval-narrow-ok new-int.simps unary-eval.simps(5))
  apply (metis 2 Value.distinct(1) Value.inject(1) asms(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-ists) 2 Value.simps(5) asms(1) intval-bits.simps int-
val-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 ∧ y = IntVal yb yi
proof -
  have bin-eval op x y ≠ UndefVal
    by (simp add: asms)
  then show ?thesis
    using asms apply (cases op; cases x; cases y; simp)
    using that by blast+
qed

```

```

lemma eval-bits-1-64:

```

```

[m,p] ⊢ xe ↦ (IntVal b ix) ⇒ 0 < b ∧ b ≤ 64
proof (induction xe arbitrary: b ix)
  case (UnaryExpr op x2)
  then obtain xv where
    xv: ([m,p] ⊢ x2 ↦ xv) ∧
        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] ⊢ x ↦ xv) ∧
        ([m,p] ⊢ y ↦ yv) ∧
        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 ∈ binary-fixed-32-ops 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  $\Rightarrow$  (IRNode  $\times$  Stamp)  $\Rightarrow$  ID option where
  find-node-and-stamp g (n,s) =
    find ( $\lambda i.$  kind g i = n  $\wedge$  stamp g i = s) (sorted-list-of-set(ids g))

export-code find-node-and-stamp

```

```

fun is-preevaluated :: IRNode  $\Rightarrow$  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  $\Rightarrow$  ID  $\Rightarrow$  IRExpr  $\Rightarrow$  bool (-  $\vdash$  -  $\simeq$  - 55)
  for g where

```

```

  ConstantNode:
     $\llbracket \text{kind } g \text{ } n = \text{ConstantNode } c \rrbracket$ 
     $\implies g \vdash n \simeq (\text{ConstantExpr } c)$  |

```

```

  ParameterNode:
     $\llbracket \text{kind } g \text{ } n = \text{ParameterNode } i; \rrbracket$ 

```

$\text{stamp } g \ n = s \parallel$   
 $\implies g \vdash n \simeq (\text{ParameterExpr } i \ s) \mid$

*ConditionalNode:*

$\llbracket \text{kind } g \ n = \text{ConditionalNode } c \ t \ f;$   
 $g \vdash c \simeq ce;$   
 $g \vdash t \simeq te;$   
 $g \vdash f \simeq fe \parallel$   
 $\implies g \vdash n \simeq (\text{ConditionalExpr } ce \ te \ fe) \mid$

*AbsNode:*

$\llbracket \text{kind } g \ n = \text{AbsNode } x;$   
 $g \vdash x \simeq xe \parallel$   
 $\implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryAbs } xe) \mid$

*NotNode:*

$\llbracket \text{kind } g \ n = \text{NotNode } x;$   
 $g \vdash x \simeq xe \parallel$   
 $\implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryNot } xe) \mid$

*NegateNode:*

$\llbracket \text{kind } g \ n = \text{NegateNode } x;$   
 $g \vdash x \simeq xe \parallel$   
 $\implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryNeg } xe) \mid$

*LogicNegationNode:*

$\llbracket \text{kind } g \ n = \text{LogicNegationNode } x;$   
 $g \vdash x \simeq xe \parallel$   
 $\implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryLogicNegation } xe) \mid$

*AddNode:*

$\llbracket \text{kind } g \ n = \text{AddNode } x \ y;$   
 $g \vdash x \simeq xe;$   
 $g \vdash y \simeq ye \parallel$   
 $\implies g \vdash n \simeq (\text{BinaryExpr } \text{BinAdd } xe \ ye) \mid$

*MulNode:*

$\llbracket \text{kind } g \ n = \text{MulNode } x \ y;$   
 $g \vdash x \simeq xe;$   
 $g \vdash y \simeq ye \parallel$   
 $\implies g \vdash n \simeq (\text{BinaryExpr } \text{BinMul } xe \ ye) \mid$

*SubNode:*

$\llbracket \text{kind } g \ n = \text{SubNode } x \ y;$   
 $g \vdash x \simeq xe;$   
 $g \vdash y \simeq ye \parallel$   
 $\implies g \vdash n \simeq (\text{BinaryExpr } \text{BinSub } xe \ ye) \mid$

*AndNode:*

$\llbracket \text{kind } g \ n = \text{AndNode } x \ y;$   
   $g \vdash x \simeq xe;$   
   $g \vdash y \simeq ye \rrbracket$   
 $\implies g \vdash n \simeq (\text{BinaryExpr BinAnd } xe \ ye) \mid$

*OrNode:*

$\llbracket \text{kind } g \ n = \text{OrNode } x \ y;$   
   $g \vdash x \simeq xe;$   
   $g \vdash y \simeq ye \rrbracket$   
 $\implies g \vdash n \simeq (\text{BinaryExpr BinOr } xe \ ye) \mid$

*XorNode:*

$\llbracket \text{kind } g \ n = \text{XorNode } x \ y;$   
   $g \vdash x \simeq xe;$   
   $g \vdash y \simeq ye \rrbracket$   
 $\implies g \vdash n \simeq (\text{BinaryExpr BinXor } xe \ ye) \mid$

*ShortCircuitOrNode:*

$\llbracket \text{kind } g \ n = \text{ShortCircuitOrNode } x \ y;$   
   $g \vdash x \simeq xe;$   
   $g \vdash y \simeq ye \rrbracket$   
 $\implies g \vdash n \simeq (\text{BinaryExpr BinShortCircuitOr } xe \ ye) \mid$

*LeftShiftNode:*

$\llbracket \text{kind } g \ n = \text{LeftShiftNode } x \ y;$   
   $g \vdash x \simeq xe;$   
   $g \vdash y \simeq ye \rrbracket$   
 $\implies g \vdash n \simeq (\text{BinaryExpr BinLeftShift } xe \ ye) \mid$

*RightShiftNode:*

$\llbracket \text{kind } g \ n = \text{RightShiftNode } x \ y;$   
   $g \vdash x \simeq xe;$   
   $g \vdash y \simeq ye \rrbracket$   
 $\implies g \vdash n \simeq (\text{BinaryExpr BinRightShift } xe \ ye) \mid$

*UnsignedRightShiftNode:*

$\llbracket \text{kind } g \ n = \text{UnsignedRightShiftNode } x \ y;$   
   $g \vdash x \simeq xe;$   
   $g \vdash y \simeq ye \rrbracket$   
 $\implies g \vdash n \simeq (\text{BinaryExpr BinURightShift } xe \ ye) \mid$

*IntegerBelowNode:*

$\llbracket \text{kind } g \ n = \text{IntegerBelowNode } x \ y;$   
   $g \vdash x \simeq xe;$   
   $g \vdash y \simeq ye \rrbracket$   
 $\implies g \vdash n \simeq (\text{BinaryExpr BinIntegerBelow } xe \ ye) \mid$

*IntegerEqualsNode:*

$\llbracket \text{kind } g \ n = \text{IntegerEqualsNode } x \ y; \\ g \vdash x \simeq xe; \\ g \vdash y \simeq ye \rrbracket \\ \implies g \vdash n \simeq (\text{BinaryExpr } \text{BinIntegerEquals } xe \ ye) \mid$

*IntegerLessThanNode:*

$\llbracket \text{kind } g \ n = \text{IntegerLessThanNode } x \ y; \\ g \vdash x \simeq xe; \\ g \vdash y \simeq ye \rrbracket \\ \implies g \vdash n \simeq (\text{BinaryExpr } \text{BinIntegerLessThan } xe \ ye) \mid$

*NarrowNode:*

$\llbracket \text{kind } g \ n = \text{NarrowNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryNarrow } \text{inputBits } \text{resultBits}) \ xe) \mid$

*SignExtendNode:*

$\llbracket \text{kind } g \ n = \text{SignExtendNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ xe) \mid$

*ZeroExtendNode:*

$\llbracket \text{kind } g \ n = \text{ZeroExtendNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits}) \ xe) \mid$

*LeafNode:*

$\llbracket \text{is-preevaluated } (\text{kind } g \ n); \\ \text{stamp } g \ n = s \rrbracket \\ \implies g \vdash n \simeq (\text{LeafExpr } n \ s) \mid$

*RefNode:*

$\llbracket \text{kind } g \ n = \text{RefNode } n'; \\ g \vdash n' \simeq e \rrbracket \\ \implies g \vdash n \simeq e$

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

**inductive**

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

*RepNil*:

$g \vdash [] \simeq_L [] \mid$

*RepCons*:  
 $\llbracket g \vdash x \simeq xe; \\ g \vdash xs \simeq_L xse \rrbracket \\ \implies g \vdash x \# xs \simeq_L xe \# xse$

**code-pred** (*modes*:  $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$  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*  $\Rightarrow$  *ID*  $\Rightarrow$  *ID*  $\Rightarrow$  *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*  $\Rightarrow$  *ID*  $\Rightarrow$  *bool* **where**  
 $n \notin \text{ids } g \implies \text{fresh-id } g \ n$

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



```

fun get-fresh-id :: IRGraph  $\Rightarrow$  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  $\Rightarrow$  IRExpr  $\Rightarrow$  (IRGraph  $\times$  ID)  $\Rightarrow$  bool (-  $\oplus$  -  $\rightsquigarrow$  - 55)
  where

    ConstantNodeSame:
     $\llbracket \text{find-node-and-stamp } g \text{ (ConstantNode } c, \text{ constantAsStamp } c) = \text{Some } n \rrbracket$ 
       $\implies g \oplus (\text{ConstantExpr } c) \rightsquigarrow (g, n) \mid$ 

    ConstantNodeNew:
     $\llbracket \text{find-node-and-stamp } g \text{ (ConstantNode } c, \text{ constantAsStamp } c) = \text{None};$ 
       $n = \text{get-fresh-id } g;$ 
       $g' = \text{add-node } n \text{ (ConstantNode } c, \text{ constantAsStamp } c) \text{ } g \rrbracket$ 
       $\implies g \oplus (\text{ConstantExpr } c) \rightsquigarrow (g', n) \mid$ 

    ParameterNodeSame:
     $\llbracket \text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{Some } n \rrbracket$ 
       $\implies g \oplus (\text{ParameterExpr } i \text{ } s) \rightsquigarrow (g, n) \mid$ 

    ParameterNodeNew:
     $\llbracket \text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{None};$ 
       $n = \text{get-fresh-id } g;$ 
       $g' = \text{add-node } n \text{ (ParameterNode } i, s) \text{ } g \rrbracket$ 
       $\implies g \oplus (\text{ParameterExpr } i \text{ } s) \rightsquigarrow (g', n) \mid$ 

    ConditionalNodeSame:
     $\llbracket g \oplus ce \rightsquigarrow (g2, c);$ 
       $g2 \oplus te \rightsquigarrow (g3, t);$ 
       $g3 \oplus fe \rightsquigarrow (g4, f);$ 
       $s' = \text{meet (stamp } g4 \text{ } t) \text{ (stamp } g4 \text{ } f);$ 
       $\text{find-node-and-stamp } g4 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{Some } n \rrbracket$ 
       $\implies g \oplus (\text{ConditionalExpr } ce \text{ } te \text{ } fe) \rightsquigarrow (g4, n) \mid$ 

    ConditionalNodeNew:
     $\llbracket g \oplus ce \rightsquigarrow (g2, c);$ 
       $g2 \oplus te \rightsquigarrow (g3, t);$ 
       $g3 \oplus fe \rightsquigarrow (g4, f);$ 
       $s' = \text{meet (stamp } g4 \text{ } t) \text{ (stamp } g4 \text{ } f);$ 
       $\text{find-node-and-stamp } g4 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{None};$ 

```

$n = \text{get-fresh-id } g4;$   
 $g' = \text{add-node } n \text{ (ConditionalNode } c \text{ } t \text{ } f, s') \text{ } g4$   
 $\implies g \oplus (\text{ConditionalExpr } ce \text{ } te \text{ } fe) \rightsquigarrow (g', n) \mid$

*UnaryNodeSame:*

$\llbracket g \oplus xe \rightsquigarrow (g2, x);$   
 $s' = \text{stamp-unary op (stamp } g2 \text{ } x);$   
 $\text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{Some } n \rrbracket$   
 $\implies g \oplus (\text{UnaryExpr op } xe) \rightsquigarrow (g2, n) \mid$

*UnaryNodeNew:*

$\llbracket g \oplus xe \rightsquigarrow (g2, x);$   
 $s' = \text{stamp-unary op (stamp } g2 \text{ } x);$   
 $\text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{None};$   
 $n = \text{get-fresh-id } g2;$   
 $g' = \text{add-node } n \text{ (unary-node op } x, s') \text{ } g2$   
 $\implies g \oplus (\text{UnaryExpr op } xe) \rightsquigarrow (g', n) \mid$

*BinaryNodeSame:*

$\llbracket g \oplus xe \rightsquigarrow (g2, x);$   
 $g2 \oplus ye \rightsquigarrow (g3, y);$   
 $s' = \text{stamp-binary op (stamp } g3 \text{ } x) \text{ (stamp } g3 \text{ } y);$   
 $\text{find-node-and-stamp } g3 \text{ (bin-node op } x \text{ } y, s') = \text{Some } n \rrbracket$   
 $\implies g \oplus (\text{BinaryExpr op } xe \text{ } ye) \rightsquigarrow (g3, n) \mid$

*BinaryNodeNew:*

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

*AllLeafNodes:*

$\llbracket \text{stamp } g \text{ } n = s;$   
 $\text{is-preevaluated (kind } g \text{ } n) \rrbracket$   
 $\implies g \oplus (\text{LeafExpr } n \text{ } s) \rightsquigarrow (g, n)$

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

$$\begin{array}{c}
\frac{\text{find-node-and-stamp } g \text{ (ConstantNode } c, \text{ constantAsStamp } c) = \text{Some } n}{g \oplus \text{ConstantExpr } c \rightsquigarrow (g, n)} \\
\\
\frac{\begin{array}{c} \text{find-node-and-stamp } g \text{ (ConstantNode } c, \text{ constantAsStamp } c) = \text{None} \\ n = \text{get-fresh-id } g \\ g' = \text{add-node } n \text{ (ConstantNode } c, \text{ constantAsStamp } c) \end{array}}{g \oplus \text{ConstantExpr } c \rightsquigarrow (g', n)} \\
\\
\frac{\text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{Some } n}{g \oplus \text{ParameterExpr } i \text{ } s \rightsquigarrow (g, n)} \\
\\
\frac{\begin{array}{c} \text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{None} \\ n = \text{get-fresh-id } g \quad g' = \text{add-node } n \text{ (ParameterNode } i, s) \end{array}}{g \oplus \text{ParameterExpr } i \text{ } s \rightsquigarrow (g', n)} \\
\\
\frac{\begin{array}{c} g \oplus ce \rightsquigarrow (g2, c) \quad g2 \oplus te \rightsquigarrow (g3, t) \\ g3 \oplus fe \rightsquigarrow (g4, f) \quad s' = \text{meet (stamp } g4 \text{ } t) \text{ (stamp } g4 \text{ } f) \\ \text{find-node-and-stamp } g4 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{Some } n \end{array}}{g \oplus \text{ConditionalExpr } ce \text{ } te \text{ } fe \rightsquigarrow (g4, n)} \\
\\
\frac{\begin{array}{c} g \oplus ce \rightsquigarrow (g2, c) \quad g2 \oplus te \rightsquigarrow (g3, t) \\ g3 \oplus fe \rightsquigarrow (g4, f) \quad s' = \text{meet (stamp } g4 \text{ } t) \text{ (stamp } g4 \text{ } f) \\ \text{find-node-and-stamp } g4 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{None} \\ n = \text{get-fresh-id } g4 \quad g' = \text{add-node } n \text{ (ConditionalNode } c \text{ } t \text{ } f, s') \end{array}}{g \oplus \text{ConditionalExpr } ce \text{ } te \text{ } fe \rightsquigarrow (g', n)} \\
\\
\frac{\begin{array}{c} g \oplus xe \rightsquigarrow (g2, x) \\ g2 \oplus ye \rightsquigarrow (g3, y) \quad s' = \text{stamp-binary op (stamp } g3 \text{ } x) \text{ (stamp } g3 \text{ } y) \\ \text{find-node-and-stamp } g3 \text{ (bin-node op } x \text{ } y, s') = \text{Some } n \end{array}}{g \oplus \text{BinaryExpr op } xe \text{ } ye \rightsquigarrow (g3, n)} \\
\\
\frac{\begin{array}{c} g \oplus xe \rightsquigarrow (g2, x) \\ g2 \oplus ye \rightsquigarrow (g3, y) \quad s' = \text{stamp-binary op (stamp } g3 \text{ } x) \text{ (stamp } g3 \text{ } y) \\ \text{find-node-and-stamp } g3 \text{ (bin-node op } x \text{ } y, s') = \text{None} \\ n = \text{get-fresh-id } g3 \quad g' = \text{add-node } n \text{ (bin-node op } x \text{ } y, s') \end{array}}{g \oplus \text{BinaryExpr op } xe \text{ } ye \rightsquigarrow (g', n)} \\
\\
\frac{\begin{array}{c} g \oplus xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op (stamp } g2 \text{ } x) \\ \text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{Some } n \end{array}}{g \oplus \text{UnaryExpr op } xe \rightsquigarrow (g2, n)} \\
\\
\frac{\begin{array}{c} g \oplus xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op (stamp } g2 \text{ } x) \\ \text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{None} \\ n = \text{get-fresh-id } g2 \quad g' = \text{add-node } n \text{ (unary-node op } x, s') \end{array}}{g \oplus \text{UnaryExpr op } xe \rightsquigarrow (g', n)} \\
\\
\frac{\text{stamp } g \text{ } n = s \quad \text{is-preevaluated (kind } g \text{ } n)}{g \oplus \text{LeafExpr } n \text{ } s \rightsquigarrow (g, n)}
\end{array}$$

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

## 2.3 Lift Data-flow Tree Semantics

**definition** *encodeeval* :: *IRGraph*  $\Rightarrow$  *MapState*  $\Rightarrow$  *Params*  $\Rightarrow$  *ID*  $\Rightarrow$  *Value*  $\Rightarrow$  *bool*  
 $([-, -, ] \vdash - \mapsto - \ 50)$   
**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*  $\Rightarrow$  *ID*  $\Rightarrow$  *IRExpr*  $\Rightarrow$  *bool*  
 $(- \vdash - \preceq - \ 50)$   
**where**  
 $(g \vdash n \preceq e) = (\exists e'. (g \vdash n \simeq e') \wedge (e' \leq e))$

**definition** *graph-refinement* :: *IRGraph*  $\Rightarrow$  *IRGraph*  $\Rightarrow$  *bool* **where**  
*graph-refinement* *g1 g2* =  
 $((ids\ g_1 \subseteq ids\ g_2) \wedge$   
 $(\forall n . n \in ids\ g_1 \longrightarrow (\forall e. (g_1 \vdash n \simeq e) \longrightarrow (g_2 \vdash n \preceq e))))$

**lemma** *graph-refinement*:

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

## 2.5 Maximal Sharing

**definition** *maximal-sharing*:  
*maximal-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**