

Veriopt Theories

August 25, 2022

Contents

1	Data-flow Semantics	1
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	Data-flow Tree Theorems	7
1.5.1	Deterministic Data-flow Evaluation	7
1.5.2	Typing Properties for Integer Evaluation Functions . .	7
1.5.3	Evaluation Results are Valid	8
1.5.4	Example Data-flow Optimisations	11
1.5.5	Monotonicity of Expression Refinement	12
1.6	Unfolding rules for evaltree quadruples down to bin-eval level	13
2	Tree to Graph	14
2.1	Subgraph to Data-flow Tree	14
2.2	Data-flow Tree to Subgraph	18
2.3	Lift Data-flow Tree Semantics	23
2.4	Graph Refinement	23
2.5	Maximal Sharing	23

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: IRExp) (ir-y: IRExp)
| ConditionalExpr (ir-condition: IRExp) (ir-trueValue: IRExp) (ir-falseValue:
IRExp)

| 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 :: IRExp ⇒ 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 :: IRExp . is-ground e }
using is-ground.simps(6) by blast

```

1.2 Functions for re-calculating stamps

Note: all integer calculations are done as 32 or 64 bit calculations. Most operators have the same output bits as their inputs. But the following *fixed32* 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) =
    (if op ∈ normal-unary
     then unrestricted-stamp (IntegerStamp (if b=64 then 64 else 32) lo hi)
     else unrestricted-stamp (IntegerStamp (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 ∧ b1=64
      then unrestricted-stamp (IntegerStamp 64 lo1 hi1)
      else unrestricted-stamp (IntegerStamp 32 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 out-
Bits v |
  unary-eval (UnaryZeroExtend inBits outBits) v = intval-zero-extend inBits out-
Bits 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* \Rightarrow *Value* \Rightarrow *bool* **where**
 $\llbracket \text{value} \neq \text{UndefVal} \rrbracket \implies \text{not-undef-or-fail value value}$

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

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

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

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

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

$result \neq \text{UndefVal}$
 $\implies [m,p] \vdash (\text{BinaryExpr } op \ xe \ ye) \mapsto result \mid$

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

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

inductive
evaltrees :: $\text{MapState} \Rightarrow \text{Params} \Rightarrow \text{IRExpr list} \Rightarrow \text{Value list} \Rightarrow \text{bool}$ ($[-, -] \vdash - \mapsto_L$
 - 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*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ as *evalTs*)
evaltrees .

definition *sq-param0* :: *IRExpr* **where**
 $sq-param0 = \text{BinaryExpr BinMul}$
 $(\text{ParameterExpr } 0 \ (\text{IntegerStamp } 32 \ (- \ 2147483648) \ 2147483647))$
 $(\text{ParameterExpr } 0 \ (\text{IntegerStamp } 32 \ (- \ 2147483648) \ 2147483647))$

values $\{v. \text{evaltree new-map-state } [\text{IntVal32 } 5] \ sq-param0 \ v\}$

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]:
     $(e_1 < e_2) \longleftrightarrow (e_1 \leq e_2 \wedge \neg (e_1 \dot{=} e_2))$ 
```

```
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  $x \leq y \implies y \leq z \implies x \leq z$  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
```

```
    IRTreeEval
```

```
begin
```

1.5.1 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*

1.5.2 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 *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 *isIntVal* (*unary-eval op x*)
unfolding *isIntVal-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 *isIntVal* (*bin-eval op x y*)
apply (*cases op; cases x; cases y*)
unfolding *isIntVal-def* **using** *def* **apply** *auto*
by (*metis (full-types) bool-to-val.simps isIntVal32-def*)**+**

lemma *int-stamp32*:

assumes *i: isIntVal32 v*
shows *isIntegerStamp* (*constantAsStamp v*)
using *i* **unfolding** *isIntegerStamp-def isIntVal32-def* **by** *auto*

lemma *int-stamp64*:

assumes *i: isIntVal64 v*
shows *isIntegerStamp* (*constantAsStamp v*)
using *i* **unfolding** *isIntegerStamp-def isIntVal64-def* **by** *auto*

lemma *int-stamp-both*:

assumes *i: isIntVal v*


```

shows is-IntegerStamp (constantAsStamp v)
using i unfolding is-IntVal-def is-IntegerStamp-def
using int-stamp32 int-stamp64 is-IntegerStamp-def by auto

```

```

lemma validDefIntConst:
  assumes v ≠ UndefVal
  assumes is-IntegerStamp (constantAsStamp v)
  shows valid-value v (constantAsStamp v)
  using assms by (cases v; auto)

```

```

lemma validIntConst:
  assumes i: is-IntVal v
  shows valid-value v (constantAsStamp v)
  using i int-stamp-both is-IntVal-def validDefIntConst by auto

```

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-int1[elim]:
  shows valid-value val (IntegerStamp 1 lo hi) ⇒
    (∃ v. val = IntVal32 v)
  apply (rule val-to-bool.cases[of val])
  using Value.distinct by simp+

```

```

lemma valid-int8[elim]:
  shows valid-value val (IntegerStamp 8 l h) ⇒
    (∃ v. val = IntVal32 v)
  apply (rule val-to-bool.cases[of val])
  using Value.distinct by simp+

```

```

lemma valid-int16[elim]:

```

```

shows valid-value val (IntegerStamp 16 l h) ==>
  ( $\exists v. \text{val} = \text{IntVal32 } v$ )
apply (rule val-to-bool.cases[of val])
using Value.distinct by simp+

lemma valid-int32[elim]:
shows valid-value val (IntegerStamp 32 l h) ==>
  ( $\exists v. \text{val} = \text{IntVal32 } v$ )
apply (rule val-to-bool.cases[of val])
using Value.distinct by simp+

lemma valid-int64[elim]:
shows valid-value val (IntegerStamp 64 l h) ==>
  ( $\exists v. \text{val} = \text{IntVal64 } v$ )
apply (rule val-to-bool.cases[of val])
using Value.distinct by simp+

lemmas valid-value-elim =
  valid-VoidStamp
  valid-ObjStamp
  valid-int1
  valid-int8
  valid-int16
  valid-int32
  valid-int64

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

lemma leafint32:
assumes ev: [m,p] ⊢ LeafExpr i (IntegerStamp 32 lo hi) ↦ val
shows  $\exists v. \text{val} = (\text{IntVal32 } v)$ 

proof –
  have valid-value val (IntegerStamp 32 lo hi)
    using ev by (rule LeafExprE; simp)
  then show ?thesis by auto
qed

lemma leafint64:
assumes ev: [m,p] ⊢ LeafExpr i (IntegerStamp 64 lo hi) ↦ val
shows  $\exists v. \text{val} = (\text{IntVal64 } v)$ 

```

```

proof –
  have valid-value val (IntegerStamp 64 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 valid32 [simp]:
  assumes valid-value val (IntegerStamp 32 lo hi)
  shows  $\exists v. (val = (IntVal32\ v) \wedge lo \leq sint\ v \wedge sint\ v \leq hi)$ 
  using assms valid-int32 by force

lemma valid64 [simp]:
  assumes valid-value val (IntegerStamp 64 lo hi)
  shows  $\exists v. (val = (IntVal64\ v) \wedge lo \leq sint\ v \wedge sint\ v \leq hi)$ 
  using assms valid-int64 by force

lemma valid32or64:
  assumes valid-value x (IntegerStamp b lo hi)
  shows  $(\exists v1. (x = IntVal32\ v1)) \vee (\exists v2. (x = IntVal64\ v2))$ 
  using valid32 valid64 assms valid-value.elims(2) by blast

lemma valid32or64-both:
  assumes valid-value x (IntegerStamp b lox hix)
  and valid-value y (IntegerStamp b loy hiy)
  shows  $(\exists v1\ v2. x = IntVal32\ v1 \wedge y = IntVal32\ v2) \vee (\exists v3\ v4. x = IntVal64\ v3 \wedge y = IntVal64\ v4)$ 
  using assms valid32or64 valid32 by (metis valid-int64 valid-value.simps(2))



### 1.5.4 Example Data-flow Optimisations

lemma a0a-helper [simp]:
  assumes a: valid-value v (IntegerStamp 32 lo hi)
  shows intval-add v (IntVal32 0) = v
proof –
  obtain v32 :: int32 where v = (IntVal32 v32) using a valid32 by blast
  then show ?thesis by simp
qed

lemma a0a: (BinaryExpr BinAdd (LeafExpr 1 default-stamp) (ConstantExpr (IntVal32 0)))
   $\geq$  (LeafExpr 1 default-stamp)
  by (auto simp add: evaltree.LeafExpr)

```

```

lemma xyx-y-helper [simp]:
  assumes valid-value x (IntegerStamp 32 lox hix)
  assumes valid-value y (IntegerStamp 32 loy hiy)
  shows intval-add x (intval-sub y x) = y
proof –
  obtain x32 :: int32 where x: x = (IntVal32 x32) using assms valid32 by blast
  obtain y32 :: int32 where y: y = (IntVal32 y32) using assms valid32 by blast
  show ?thesis using x y by simp
qed

```

```

lemma xyx-y:
  (BinaryExpr BinAdd
    (LeafExpr x (IntegerStamp 32 lox hix))
    (BinaryExpr BinSub
      (LeafExpr y (IntegerStamp 32 loy hiy))
      (LeafExpr x (IntegerStamp 32 lox hix))))
  ≥ (LeafExpr y (IntegerStamp 32 loy hiy))
by (auto simp add: LeafExpr)

```

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

```

lemma mono-unary:
  assumes e ≥ e'
  shows (UnaryExpr op e) ≥ (UnaryExpr op e')
  using UnaryExpr assms by auto

```

```

lemma mono-binary:
  assumes x ≥ x'
  assumes y ≥ y'
  shows (BinaryExpr op x y) ≥ (BinaryExpr op x' y')
  using BinaryExpr assms by auto

```

```

lemma never-void:
  assumes [m, p] ⊢ x ↦ xv
  assumes valid-value xv (stamp-expr xe)
  shows stamp-expr xe ≠ 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* (*verit*, *best*) *compatible.elims*(2) *compatible.simps*(1))

lemma *compatible-refl*:
 $compatible\ x\ y \implies compatible\ y\ x$
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'\ te'\ fe'*
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-valid32* [*simp*]:
 $valid\text{-}value\ y\ (constantAsStamp\ (IntVal32\ v)) = (y = IntVal32\ v)$
by (*induction* y ; *auto* *dest*: *signed-word-eqI*)

lemma *unfold-valid64* [*simp*]:
 $valid\text{-}value\ y\ (constantAsStamp\ (IntVal64\ v)) = (y = IntVal64\ v)$
by (*induction* y ; *auto* *dest*: *signed-word-eqI*)

lemma *unfold-const*:

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

corollary *unfold-const32*:

shows $([m,p] \vdash \text{ConstantExpr } (\text{IntVal32 } c) \mapsto v) = (v = \text{IntVal32 } c)$
using *unfold-valid32* **by** *blast*

corollary *unfold-const64*:

shows $([m,p] \vdash \text{ConstantExpr } (\text{IntVal64 } c) \mapsto v) = (v = \text{IntVal64 } c)$
using *unfold-valid64* **by** *blast*

lemma *unfold-binary*:

shows $([m,p] \vdash \text{BinaryExpr } op \ xe \ ye \mapsto val) = (\exists \ x \ y. \\$
 $(([m,p] \vdash xe \mapsto x) \wedge$
 $([m,p] \vdash ye \mapsto y) \wedge$
 $(val = \text{bin-eval } op \ x \ y) \wedge$
 $(val \neq \text{UndefVal})$
 $)) \text{ (is } ?L = ?R)$

proof (*intro iffI*)

assume $?L$

show $?R$ **by** (*rule evaltree.cases[OF ?L]*; *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 $([m,p] \vdash \text{UnaryExpr } op \ xe \mapsto val) \\$
 $= (\exists \ x. \\$
 $(([m,p] \vdash xe \mapsto x) \wedge$
 $(val = \text{unary-eval } op \ x) \wedge$
 $(val \neq \text{UndefVal})$
 $)) \text{ (is } ?L = ?R)$

by *auto*

lemmas *unfold-evaltree* =

unfold-binary

unfold-unary

```

    unfold-const32
    unfold-const64
    unfold-valid32
    unfold-valid64

```

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

```

```

  ParameterNode:
   $\llbracket$  kind g n = ParameterNode i;
    stamp g n = s  $\rrbracket$ 
     $\implies$  g  $\vdash$  n  $\simeq$  (ParameterExpr i s) |

```

ConditionalNode:

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

AbsNode:

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

NotNode:

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

NegateNode:

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

LogicNegationNode:

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

AddNode:

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

MulNode:

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

SubNode:

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

AndNode:

$\llbracket \text{kind } g \ n = \text{AndNode } x \ y; \\$

$g \vdash x \simeq xe;$
 $g \vdash y \simeq ye]$
 $\implies g \vdash n \simeq (\text{BinaryExpr BinAnd } xe \ ye) \mid$

OrNode:

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

XorNode:

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

ShortCircuitOrNode:

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

LeftShiftNode:

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

RightShiftNode:

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

UnsignedRightShiftNode:

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

IntegerBelowNode:

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

IntegerEqualsNode:

$\llbracket \text{kind } g \ n = \text{IntegerEqualsNode } x \ y;$
 $g \vdash x \simeq xe;$

$g \vdash y \simeq ye$
 $\implies g \vdash n \simeq (\text{BinaryExpr BinIntegerEquals } xe \ ye) \mid$

IntegerLessThanNode:

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

NarrowNode:

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

SignExtendNode:

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

ZeroExtendNode:

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

LeafNode:

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

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

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

RepNil:

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

RepCons:

$\llbracket g \vdash x \simeq xe;$

$$\begin{aligned}
& g \vdash xs \simeq_L xse \\
& \implies g \vdash x\#xs \simeq_L xe\#xse
\end{aligned}$$

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

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*

fun *choose-32-64* :: *int* \Rightarrow *int64* \Rightarrow *Value* **where**

choose-32-64 *bits val* =
 (if *bits* = 32
 then (*IntVal32* (*ucast* *val*))
 else (*IntVal64* (*val*)))

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

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

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

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

    ConditionalNodeNew:
     $\llbracket g \oplus ce \rightsquigarrow (g2, c);$ 

```

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

UnaryNodeSame:

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

UnaryNodeNew:

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

BinaryNodeSame:

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

BinaryNodeNew:

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

AllLeafNodes:

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

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*
 $(- \vdash - \sqsubseteq - \ 50)$
where
 $(g \vdash n \sqsubseteq e) = (\exists e'. (g \vdash n \simeq e') \wedge (e' \leq e))$

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

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