Veriopt Theories

July 13, 2022

Contents

1 Data-flow Semantics

theory *IRTreeEval*
  imports
    Graph.Stamp
begin

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

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

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

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

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

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

definition *new-map-state* :: *MapState* **where**
new-map-state = (λ*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: all integer calculations are done as 32 or 64 bit calculations. Most operators have the same output bits as their inputs. But the following $fixed_32$ binary operators always output 32 bits. And the unary operators that are not $normal_unary$ are narrowing or widening operators, so the result bits is specified by the operator.

abbreviation *fixed-32* :: *IRBinaryOp set* **where**
fixed-32 ≡ {*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 outBits v* |
unary-eval (*UnaryZeroExtend inBits outBits*) *v* = *intval-zero-extend inBits outBits v*

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

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

inductive *not-undef-or-fail* :: *Value* $\Rightarrow$ *Value* $\Rightarrow$ *bool* **where**
$[\![$*value* $\neq$ *UndefVal*$]\!]$ $\Longrightarrow$ *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:
$[\![$*valid-value c* (*constantAsStamp c*)$]\!]$
$\Longrightarrow$ [*m*,*p*] $\vdash$ (*ConstantExpr c*) $\mapsto$ *c* |

ParameterExpr:
$[\![$*i* < *length p*; *valid-value* (*p*!*i*) *s*$]\!]$
$\Longrightarrow$ [*m*,*p*] $\vdash$ (*ParameterExpr i s*) $\mapsto$ *p*!*i* |

ConditionalExpr:
$[\![$[*m*,*p*] $\vdash$ *ce* $\mapsto$ *cond*;
branch = (*if val-to-bool cond then te else fe*);
[*m*,*p*] $\vdash$ *branch* $\mapsto$ *v*;
v $\neq$ *UndefVal*$]\!]$
$\Longrightarrow$ [*m*,*p*] $\vdash$ (*ConditionalExpr ce te fe*) $\mapsto$ *v* |

UnaryExpr:
$[\![$[*m*,*p*] $\vdash$ *xe* $\mapsto$ *v*;
result = (*unary-eval op v*);
result $\neq$ *UndefVal*$]\!]$
$\Longrightarrow$ [*m*,*p*] $\vdash$ (*UnaryExpr op xe*) $\mapsto$ *result* |

BinaryExpr:
$[\![$[*m*,*p*] $\vdash$ *xe* $\mapsto$ *x*;
[*m*,*p*] $\vdash$ *ye* $\mapsto$ *y*;
result = (*bin-eval op x y*);
result $\neq$ *UndefVal*$]\!]$

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

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

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

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

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

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

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

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

values {*v*. *evaltree new-map-state* [*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*]:
    $(e_1 < e_2) \longleftrightarrow (e_1 \leq e_2 \wedge \neg\ (e_1 \doteq e_2))$

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

end

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

end

1.5 Data-flow Tree Theorems

theory *IRTreeEvalThms*
  imports
    IRTreeEval
begin

1.5.1 Deterministic Data-flow Evaluation

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

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

1.5.2 Typing Properties for Integer Evaluation Functions

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

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

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

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

lemma *bin-eval-int*:
  assumes *def*: *bin-eval op x y* $\neq$ *UndefVal*
  shows *is-IntVal* (*bin-eval op x y*)
  apply (*cases op*; *cases x*; *cases y*)
  unfolding *is-IntVal-def* **using** *def* **apply** *auto*
  by (*metis* (*full-types*) *bool-to-val.simps is-IntVal32-def*)+

lemma *int-stamp32*:
  assumes *i*: *is-IntVal32 v*
  shows *is-IntegerStamp* (*constantAsStamp v*)
  using *i* **unfolding** *is-IntegerStamp-def is-IntVal32-def* **by** *auto*

lemma *int-stamp64*:
  assumes *i*: *is-IntVal64 v*
  shows *is-IntegerStamp* (*constantAsStamp v*)
  using *i* **unfolding** *is-IntegerStamp-def is-IntVal64-def* **by** *auto*

lemma *int-stamp-both*:
  assumes *i*: *is-IntVal 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 $v \neq$ *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*: $s \neq$ *VoidStamp*
shows *val* $\neq$ *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* $\Longrightarrow$
val = *UndefVal*
using *valid-value.simps* **by** *metis*

lemma *valid-ObjStamp*[*elim*]:
shows *valid-value val* (*ObjectStamp klass exact nonNull alwaysNull*) $\Longrightarrow$
($\exists$ *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*) $\Longrightarrow$
($\exists$ *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*) $\Longrightarrow$
($\exists$ *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*) $\Longrightarrow$

$(\exists\, v.\ val = IntVal32\ v)$
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*) $\Longrightarrow$
$(\exists\, v.\ val = IntVal32\ v)$
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*) $\Longrightarrow$
$(\exists\, v.\ val = IntVal64\ v)$
apply (*rule val-to-bool.cases*[*of val*])
using *Value.distinct* **by** *simp+*

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

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

lemma *leafint32*:
assumes *ev*: $[m,p] \vdash LeafExpr\ i\ (IntegerStamp\ 32\ lo\ hi) \mapsto val$
shows $\exists\, v.\ val = (IntVal32\ v)$

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*: $[m,p] \vdash LeafExpr\ i\ (IntegerStamp\ 64\ lo\ hi) \mapsto val$
shows $\exists\, v.\ val = (IntVal64\ v)$

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 $mono(UnaryExprop)$, but it is not obvious how to do this for both arguments of the binary expressions.

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

lemma *mono-binary*:
assumes $x \geq x'$
assumes $y \geq y'$
shows (*BinaryExpr op x y*) ≥ (*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* ≠ *VoidStamp*
using *valid-value.simps*
using *assms*(*2*) **by** *force*

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

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

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

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

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

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

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

lemma *unfold-valid32* [*simp*]:
  valid-value y (*constantAsStamp* (*IntVal32 v*)) = (*y* = *IntVal32 v*)
  by (*induction y*; *auto dest*: *signed-word-eqI*)

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

lemma *unfold-const*:

shows ([*m*,*p*] ⊢ *ConstantExpr c* ↦ *v*) = (*valid-value v* (*constantAsStamp c*) ∧ *v* = *c*)
by *blast*

corollary *unfold-const32*:
shows ([*m*,*p*] ⊢ *ConstantExpr* (*IntVal32 c*) ↦ *v*) = (*v* = *IntVal32 c*)
using *unfold-valid32* **by** *blast*

corollary *unfold-const64*:
shows ([*m*,*p*] ⊢ *ConstantExpr* (*IntVal64 c*) ↦ *v*) = (*v* = *IntVal64 c*)
using *unfold-valid64* **by** *blast*

lemma *unfold-binary*:
shows ([*m*,*p*] ⊢ *BinaryExpr op xe ye* ↦ *val*) = (∃ *x y*.
(([*m*,*p*] ⊢ *xe* ↦ *x*) ∧
([*m*,*p*] ⊢ *ye* ↦ *y*) ∧
(*val* = *bin-eval op x y*) ∧
(*val* ≠ *UndefVal*)
)) (**is** *?L* = *?R*)
proof (*intro iffI*)
assume *3*: *?L*
show *?R* **by** (*rule evaltree.cases*[*OF 3*]; *blast+*)
next
assume *?R*
then obtain *x y* **where** [*m*,*p*] ⊢ *xe* ↦ *x*
and [*m*,*p*] ⊢ *ye* ↦ *y*
and *val* = *bin-eval op x y*
and *val* ≠ *UndefVal*
by *auto*
then show *?L*
by (*rule BinaryExpr*)
qed

lemma *unfold-unary*:
shows ([*m*,*p*] ⊢ *UnaryExpr op xe* ↦ *val*)
= (∃ *x*.
(([*m*,*p*] ⊢ *xe* ↦ *x*) ∧
(*val* = *unary-eval op x*) ∧
(*val* ≠ *UndefVal*)
)) (**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* ⇒ (*IRNode* × *Stamp*) ⇒ *ID option* **where**
find-node-and-stamp g (*n*,*s*) =
find (λ*i*. *kind g i* = *n* ∧ *stamp g i* = *s*) (*sorted-list-of-set*(*ids g*))

export-code *find-node-and-stamp*

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

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

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

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

ConditionalNode:

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

$AbsNode$:
$\llbracket kind\ g\ n = AbsNode\ x;$
$\ \ g \vdash x \simeq xe\rrbracket$
$\ \ \Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryAbs\ xe) \mid$

$NotNode$:
$\llbracket kind\ g\ n = NotNode\ x;$
$\ \ g \vdash x \simeq xe\rrbracket$
$\ \ \Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryNot\ xe) \mid$

$NegateNode$:
$\llbracket kind\ g\ n = NegateNode\ x;$
$\ \ g \vdash x \simeq xe\rrbracket$
$\ \ \Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryNeg\ xe) \mid$

$LogicNegationNode$:
$\llbracket kind\ g\ n = LogicNegationNode\ x;$
$\ \ g \vdash x \simeq xe\rrbracket$
$\ \ \Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryLogicNegation\ xe) \mid$

$AddNode$:
$\llbracket kind\ g\ n = AddNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye\rrbracket$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinAdd\ xe\ ye) \mid$

$MulNode$:
$\llbracket kind\ g\ n = MulNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye\rrbracket$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinMul\ xe\ ye) \mid$

$SubNode$:
$\llbracket kind\ g\ n = SubNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$
$\ \ g \vdash y \simeq ye\rrbracket$
$\ \ \Longrightarrow g \vdash n \simeq (BinaryExpr\ BinSub\ xe\ ye) \mid$

$AndNode$:
$\llbracket kind\ g\ n = AndNode\ x\ y;$
$\ \ g \vdash x \simeq xe;$

$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinAnd\ xe\ ye)\ |$

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

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

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

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

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

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

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

$IntegerEqualsNode$:
$[\![kind\ g\ n = IntegerEqualsNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$

$\Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinIntegerEquals}\ xe\ ye) \mid$

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

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

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

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

LeafNode:
$[\![\mathit{is\text{-}preevaluated}\ (\mathit{kind}\ g\ n);$
$\mathit{stamp}\ g\ n = s]\!]$
$\Longrightarrow g \vdash n \simeq (\mathit{LeafExpr}\ n\ s) \mid$

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

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

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

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

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

$\Longrightarrow g \vdash x\#xs \simeq_L xe\#xse$

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

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

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

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**
$n \notin ids\ g \Longrightarrow fresh\text{-}id\ g\ n$

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

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

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

export-code *get-fresh-id*

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

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

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

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

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

ParameterNodeNew:
⟦*find-node-and-stamp g* (*ParameterNode i*, *s*) = *None*;
n = *get-fresh-id g*;
g′ = *add-node n* (*ParameterNode i*, *s*) *g*⟧
⟹ *g* ⊕ (*ParameterExpr i s*) ⇝ (*g′*, *n*) |

ConditionalNodeSame:
⟦*g* ⊕ *ce* ⇝ (*g2*, *c*);
g2 ⊕ *te* ⇝ (*g3*, *t*);
g3 ⊕ *fe* ⇝ (*g4*, *f*);
s′ = *meet* (*stamp g4 t*) (*stamp g4 f*);
find-node-and-stamp g4 (*ConditionalNode c t f*, *s′*) = *Some n*⟧
⟹ *g* ⊕ (*ConditionalExpr ce te fe*) ⇝ (*g4*, *n*) |

ConditionalNodeNew:
⟦*g* ⊕ *ce* ⇝ (*g2*, *c*);
g2 ⊕ *te* ⇝ (*g3*, *t*);

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

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

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

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

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

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

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

unrepRules

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

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

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

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

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

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

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

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

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

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

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

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

2.3 Lift Data-flow Tree Semantics

definition *encodeeval* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID* ⇒ *Value* ⇒ *bool*
  ([-,-,-] ⊢ - ↦ - *50*)
  where
  encodeeval g m p n v = (∃ *e*. (*g* ⊢ *n* ≃ *e*) ∧ ([*m*,*p*] ⊢ *e* ↦ *v*))

2.4 Graph Refinement

definition *graph-represents-expression* :: *IRGraph* ⇒ *ID* ⇒ *IRExpr* ⇒ *bool*
  (- ⊢ - ⊴ - *50*)
  where
  (*g* ⊢ *n* ⊴ *e*) = (∃ *e′* . (*g* ⊢ *n* ≃ *e′*) ∧ (*e′* ≤ *e*))

definition *graph-refinement* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
  graph-refinement g_1 g_2 =
    ((*ids* g_1 ⊆ *ids* g_2) ∧
    (∀ *n* . *n* ∈ *ids* g_1 ⟶ (∀ *e*. (g_1 ⊢ *n* ≃ *e*) ⟶ (g_2 ⊢ *n* ⊴ *e*))))

lemma *graph-refinement*:
  graph-refinement g1 g2 ⟹ (∀ *n m p v*. *n* ∈ *ids g1* ⟶ ([*g1*, *m*, *p*] ⊢ *n* ↦ *v*) ⟶ ([*g2*, *m*, *p*] ⊢ *n* ↦ *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 = (∀ n_1 n_2 . n_1 ∈ *true-ids g* ∧ n_2 ∈ *true-ids g* ⟶
    (∀ *e*. (*g* ⊢ n_1 ≃ *e*) ∧ (*g* ⊢ n_2 ≃ *e*) ∧ (*stamp g* n_1 = *stamp g* n_2) ⟶ n_1 = n_2))

end