GraalVM Canonicalization Optimizations

January 8, 2022

Abstract

This document presents the canonicalization rules which are present in the GraalVM compiler. First, individual rules are encoded in a high-level domain specific language. As these optimizations are encoded, a proof of semantics preservation is given. Next, rules are combined via a tactic language. The combined rules are then proved to be terminating. Finally, optimization phases are composed of the combined rules and generated into Java code.

Contents

```
theory CanonicalizationSyntax
imports CanonicalizationTreeProofs
keywords
  phase :: thy-decl and
  optimization :: thy-goal-defn and
  print-optimizations :: diag
begin


fun size :: IRExpr ⇒ nat where
  size (UnaryExpr op e) = (size e) + 1 |
  size (BinaryExpr BinAdd x y) = (size x) + ((size y) ∗ 2) |
  size (BinaryExpr op x y) = (size x) + (size y) |
  size (ConditionalExpr cond t f) = (size cond) + (size t) + (size f) + 2 |
  size (ConstantExpr const) = 1 |
  size (ParameterExpr ind s) = 2 |
  size (LeafExpr nid s) = 2 |
  size (ConstantVar c) = 2 |
  size (VariableExpr x s) = 2

lemma size-gt-0: size e > 0
proof (induction e)
case (UnaryExpr x1 e)
  then show ?case by auto
next
case (BinaryExpr x1 e1 e2)
then show ?case by (cases x1; auto)
next
  case (ConditionalExpr e1 e2 e3)
  then show ?case by auto
next
  case (ParameterExpr x1 x2)
then show ?case by auto
next
  case (LeafExpr x1 x2)
  then show ?case by auto
next
  case (ConstantExpr x)
  then show ?case by auto
next
  case (ConstantVar x)
  then show ?case by auto
next
  case (VariableExpr x1 x2)
  then show ?case by auto
qed

lemma binary-expr-size-gte-2: size (BinaryExpr op x y) ≥ 2
  apply (induction BinaryExpr op x y) apply auto apply (cases op; auto) using
```

size-gt-0
  apply (*metis One-nat-def Suc-leI add-le-mono mult-2-right numeral-Bit0 numeral-code*(*1*) *trans-le-add2*)
 by (*metis Suc-leI add-2-eq-Suc′ add-Suc-shift add-mono numeral-2-eq-2 size-gt-0*)+

lemma *size e* = *1* $\Longrightarrow$ *is-ConstantExpr e*
  apply (*cases e*; *auto*) **using** *size-gt-0*
  apply (*metis less-numeral-extra*(*3*)) **using** *size-gt-0*
  by (*metis binary-expr-size-gte-2 lessI not-less numeral-2-eq-2*)

lemma *nonconstants-gt-one*: ¬ (*is-ConstantExpr e*) $\Longrightarrow$ *size e* > *1*
  apply (*cases e*; *auto*) **using** *size-gt-0*
  apply *simp* **using** *size-gt-0*
  using *Suc-le-eq binary-expr-size-gte-2 numeral-2-eq-2* **by** *auto*

lemma *size-det*: *x* = *y* $\Longrightarrow$ *size x* = *size y*
  by *auto*

datatype ′*a Rewrite* =
  Transform ′*a* ′*a* |
  Conditional ′*a* ′*a bool* |
  Sequential ′*a Rewrite* ′*a Rewrite* |
  Transitive ′*a Rewrite*

ML-val ‹@{*term Transform a a*}›
ML-val ‹@{*term Conditional a b c*}›
ML-val ‹@{*term Sequential a b*}›
ML-val ‹@{*term Transitive a*}›

fun *rewrite-obligation* :: *IRExpr Rewrite* ⇒ *bool* **where**
  rewrite-obligation (*Transform x y*) = (*y* ≤ *x*) |
  rewrite-obligation (*Conditional x y cond*) = (*cond* ⟶ (*y* ≤ *x*)) |
  rewrite-obligation (*Sequential x y*) = (*rewrite-obligation x* ∧ *rewrite-obligation y*)
|
  rewrite-obligation (*Transitive x*) = *rewrite-obligation x*

ML-val ‹@{*term rewrite-obligation a*}›

ML ‹
val debugMode = *false*

fun debugPrint value =
  if debugMode then (@{*print*} *value*) *else value*

fun translateConst (*str*, *typ*) =
  case (*str*, *typ*) *of*

```
  (const Groups.plus-class.plus, -) => @{const BinaryExpr} $ @{const BinAdd}
    | (const Groups.minus-class.minus, -) => @{const BinaryExpr} $ @{const
BinSub}
    | (const Groups.times-class.times, -) => @{const BinaryExpr} $ @{const Bin-
Mul}
    | (const HOL.conj, -) => @{const BinaryExpr} $ @{const BinAnd}
    | (const -binEquals, -) => @{const BinaryExpr} $ @{const BinIntegerEquals}
    | (const Groups.uminus-class.uminus, -) => @{const UnaryExpr} $ @{const
UnaryNeg}
    | (const Values.shiftl, -) => @{const BinaryExpr} $ @{const BinLeftShift}
    | (const Values.shiftr, -) => @{const BinaryExpr} $ @{const BinRightShift}
    | (const Values.sshiftr, -) => @{const BinaryExpr} $ @{const BinURightShift}
    | - => Const (str, typ)

fun translateEquals - terms =
  @{const BinaryExpr} $ @{const BinIntegerEquals} $ hd terms $ hd (tl terms)

(* A seemingly arbitrary distinction *)
fun translateFree (str, typ) =
  case (str, typ) of
    (abs, -) => @{const UnaryExpr} $ @{const UnaryAbs}
    | (var, typ) =>
      (if String.sub(var,0) = #c
        then @{const ConstantExpr} $ Free (val- ^ var, typ)
        else Free (var, typ))

fun expandNode ctxt trm =
  let
    val - = debugPrint Expanding node;
    val - = debugPrint trm;
  in
  case trm of
    Const (str, typ) => translateConst (str, typ)
    | Free (str, typ) => translateFree (str, typ)
    | Abs (str, typ, trm) => Abs (str, typ, expandNode ctxt trm)
    | e as ((Const (const IRTreeEval.IRExpr.ConstantExpr,-)) $ -) => e
    | (x $ y) => (expandNode ctxt x $ expandNode ctxt y)
    | - => trm
  end

fun expandNodes ctxt [trm] = expandNode ctxt trm
  | expandNodes - ts = raise TERM (expandNodes, ts)

fun baseTransform ctxt [pre, post] =
  Const
      (CanonicalizationSyntax.Rewrite.Transform, @{typ IRExpr => IRExpr ⇒
IRExpr Rewrite})
    $ expandNode ctxt pre
    $ expandNode ctxt post
  | baseTransform - ts = raise TERM (baseTransform, ts)

fun conditionTransform ctxt [pre, post, cond] =
  Const (CanonicalizationSyntax.Rewrite.Conditional, @{typ IRExpr ⇒ IRExpr ⇒
bool ⇒ IRExpr Rewrite})
    $ expandNode ctxt pre
    $ expandNode ctxt post
    $ cond

  | conditionTransform - ts = raise TERM (conditionTransform, ts)

fun constantValues - [trm] =
  (case trm of
    c as Const - =>
      @{const ConstantExpr} $ (@{const IntVal32} $ c)
    | x $ y =>
      @{const ConstantExpr} $ (@{const IntVal32} $ (x $ y))
    | - => trm)
  | constantValues - ts = raise TERM (constantValues, ts)

›
```

syntax *-constantValues* :: *term* ⇒ *term* (*const* - *120*)
parse-translation ‹ [(@{*syntax-const -constantValues*} , *constantValues*)] ›

notation *ConditionalExpr* (- *?* - : -)
syntax *-binEquals* :: *term* ⇒ *term* ⇒ *term* (- == - *100*)
parse-translation ‹ [(@{*syntax-const -binEquals*} , *translateEquals*)] ›

syntax *-expandNodes* :: *term* ⇒ *term* (*exp*[-])
parse-translation ‹ [(@{*syntax-const -expandNodes*} , *expandNodes*)] ›

syntax *-baseTransform* :: *term* ⇒ *term* ⇒ *term* (- ↦ - *10*)
parse-translation ‹ [(@{*syntax-const -baseTransform*} , *baseTransform*)] ›

syntax *-conditionalTransform* :: *term* ⇒ *term* ⇒ *term* ⇒ *term* (- ↦ - *when* - *70*)
parse-translation ‹ [(@{*syntax-const -conditionalTransform*} , *conditionTransform*)] ›

value *exp*[*abs e*]
ML-val ‹@{*term abs e*}›
ML-val ‹@{*term x* & *x*}›
ML-val ‹@{*term cond ? tv* : *fv*}›
ML-val ‹@{*term* $x < y$}›
ML-val ‹@{*term* $c < y$}›
ML-val ‹@{*term* $a \Longrightarrow c < y$}›
ML-val ‹@{*term x* << *y*}›

```
value exp[c1 + y]

datatype Type =
  Integer |
  Float |
  Object |
  Unknown

definition type :: IRExpr ⇒ Type where
  type e = (case (stamp-expr e) of
    IntegerStamp - - - ⇒ Integer
    | ObjectStamp - - - - ⇒ Object
    | - ⇒ Unknown)

lemma unfold-type[simp]:
  (type x = Integer) = is-IntegerStamp (stamp-expr x)
  unfolding type-def using is-IntegerStamp-def
  using Stamp.case-eq-if Stamp.disc(1) Type.distinct(1) Type.distinct(3)
  by (simp add: Stamp.case-eq-if)

definition type-safe :: IRExpr ⇒ IRExpr ⇒ bool where
  type-safe e1 e2 =
    ((type e1 = type e2)
    ∧ (is-IntegerStamp (stamp-expr e1)
        ⟶ (stp-bits (stamp-expr e1) = stp-bits (stamp-expr e2))))

fun int-and-equal-bits :: Value ⇒ Value ⇒ bool where
  int-and-equal-bits (IntVal32 e1) (IntVal32 e2) = True |
  int-and-equal-bits (IntVal64 e1) (IntVal64 e2) = True |
  int-and-equal-bits - - = False

lemma unfold-int-typesafe[simp]:
  assumes type e1 = Integer
  shows type-safe e1 e2 =
    ((type e1 = type e2) ∧
    (stp-bits (stamp-expr e1) = stp-bits (stamp-expr e2)))
proof −
  have is-IntegerStamp (stamp-expr e1)
    using assms unfold-type by simp
  then show ?thesis unfolding type-safe-def
    by simp
qed

experiment begin
lemma add-intstamp-prop:
  assumes type x = Integer
  assumes type-safe x y
  shows type exp[x + y] = Integer
  using assms unfolding type-def type-safe-def
  using stamp-expr.simps(3) stamp-binary.simps(1)
  using is-IntegerStamp-def type-def unfold-type sorry

lemma sub-intstamp-prop:
  assumes type x = Integer
  assumes type-safe x y
  shows type exp[x − y] = Integer
  using assms unfolding type-def type-safe-def
  using stamp-expr.simps(3) stamp-binary.simps(1) sorry
end

lemma uminus-intstamp-prop:
  assumes type x = Integer
  shows type exp[−x] = Integer
  using assms unfolding type-def type-safe-def
  using stamp-expr.simps(1) stamp-unary.simps(1)
 by (metis Stamp.collapse(1) Stamp.discI(1) type-def unfold-type unrestricted-stamp.simps(2))


lemma assume-proof :
  assumes type x = Integer
  assumes type-safe x y
  shows rewrite-obligation ((x + (−y) ↦ x − y))
  unfolding rewrite-obligation.simps
  unfolding le-expr-def apply (rule allI)+ apply (rule impI)
  using assms unfolding type-def type-safe-def
  using CanonicalizeAddProof CanonicalizeAdd.intros
  sorry



lemma rewrite-obligation ((x + (−y)) ↦ (x − y) when (type x = Integer ∧
type-safe x y))
  sorry


lemma (size exp[x + (−y)]) > (size exp[x − y])
  using size.simps(1,2)
  by force



ML ‹
datatype 'a Rewrite =
  Transform of 'a ∗ 'a |
  Conditional of 'a ∗ 'a ∗ term |
  Sequential of 'a Rewrite ∗ 'a Rewrite |
  Transitive of 'a Rewrite
```

```
type rewrite =
  {name: string, rewrite: term Rewrite}

type phase =
  {name: string, rewrites: rewrite list, preconditions: term list}

type phase-store = (string list * (string -> phase option))

datatype phase-state =
  NoPhase of phase-store |
  InPhase of (string * phase-store)

signature PhaseState =
sig
  val get: theory -> phase-state
  val add: rewrite -> theory -> theory
  val reset: theory -> theory
  val enter-phase: string -> theory -> theory
  val exit-phase: theory -> theory
end;

structure RWList: PhaseState =
struct

val empty = NoPhase ([], (fn - => NONE));

fun merge-maps (left: 'a list * ('a -> 'b option)) (right: 'a list * ('a -> 'b option))
=
  ((fst left) @ (fst right), fn x => (case (snd left) x of
    NONE => (snd right) x |
    SOME v => SOME v))

structure RewriteStore = Theory-Data
(
  type T = phase-state;
  val empty = empty;
  val extend = I;
  fun merge (lhs, rhs) =
    case lhs of
      NoPhase left-store => (case rhs of
        NoPhase right-store => NoPhase (merge-maps left-store right-store) |
          InPhase (name, right-store) => InPhase (name, (merge-maps left-store
right-store))) |
      InPhase (name, left-store) => (case rhs of
       NoPhase right-store => InPhase (name, (merge-maps left-store right-store))
|
          InPhase (name, right-store) => InPhase (name, (merge-maps left-store
right-store)))
);

val get = RewriteStore.get;

fun expand-phase rewrite (phase: phase) =
 {name = (#name phase), rewrites = cons rewrite (#rewrites phase), preconditions
= (#preconditions phase)}

fun update-existing name (dom, map) rewrite =
  let
    val value = map name
  in
    case value of
      NONE => raise TERM (phase not in store, []) |
       SOME v => InPhase (name, (dom, (fn id => (if id = name then SOME
(expand-phase rewrite v) else map id))))
  end

fun add t thy = RewriteStore.map (fn state =>
  case state of
    NoPhase - => raise TERM (error, []) |
    InPhase (name, store) => update-existing name store t
  ) thy

val reset = RewriteStore.put empty;

fun new-phase name = {name = name, rewrites = ([] : rewrite list), preconditions
= ([] : term list)};

fun enter-phase name thy = RewriteStore.map (fn state =>
  case state of
    NoPhase (dom, store) => InPhase (name, ([name] @ dom, fn id => (if id =
name then SOME (new-phase name) else store id))) |
    InPhase (-, -) => raise TERM (optimization phase already established, [])
  ) thy

fun exit-phase thy = RewriteStore.map (fn state =>
  case state of
    NoPhase - => raise TERM (phase already exited, []) |
    InPhase (-, existing) => NoPhase existing
  ) thy

end;


fun term-to-rewrite term =
  case term of
     (((Const (CanonicalizationSyntax.Rewrite.Transform, -)) $ lhs) $ rhs) =>
Transform (lhs, rhs)
```

```
    | ((((Const (CanonicalizationSyntax.Rewrite.Conditional, -)) $ lhs) $ rhs) $
cond) => Conditional (lhs, rhs, cond)
    | - => raise TERM (optimization is not a rewrite, [term])

fun rewrite-to-term rewrite =
  case rewrite of
    Transform (lhs, rhs) =>
      (Const (CanonicalizationSyntax.Rewrite.Transform, @{typ 'a => 'a})) $ lhs
$ rhs
    | Conditional (lhs, rhs, cond) =>
      (Const (CanonicalizationSyntax.Rewrite.Conditional, @{typ 'a => 'a})) $ lhs
$ rhs $ cond
    | - => raise TERM (rewrite cannot be translated yet, [])

fun term-to-obligation ctxt term =
   Syntax.check-prop ctxt (@{const Trueprop} $ (@{const rewrite-obligation} $
term))

fun rewrite-to-termination rewrite =
  case rewrite of
    Transform (lhs, rhs) => (
      @{const Trueprop}
      $ (Const (Orderings.ord-class.less, @{typ nat ⇒ nat ⇒ bool})
      $ (@{const size} $ rhs) $ (@{const size} $ lhs)))
    | Conditional (lhs, rhs, condition) => (
      Const (Pure.imp, @{typ prop ⇒ prop ⇒ prop})
      $ (@{const Trueprop} $ condition)
      $ (@{const Trueprop} $ (Const (Orderings.ord-class.less, @{typ nat ⇒ nat ⇒
bool})
      $ (@{const size} $ rhs) $ (@{const size} $ lhs))))
    | - => raise TERM (rewrite termination generation not implemented, [])

fun register-optimization
  ((bind: binding, -), opt: string) ctxt =
  let
    val term = Syntax.read-term ctxt opt;

    val rewrite = term-to-rewrite term;

    val obligation = term-to-obligation ctxt term;
    val terminating = rewrite-to-termination rewrite;

    val register = RWList.add {name=Binding.print bind, rewrite=rewrite}

    fun after-qed - ctxt =
      Local-Theory.background-theory register ctxt
  in
    Proof.theorem NONE after-qed [[(obligation, []), (terminating, [])]] ctxt
  end
```

```
val parse-optimization-declaration =
  Parse-Spec.thm-name :

val - =
  Outer-Syntax.local-theory-to-proof command-keyword‹optimization›
    define an optimization and open proof obligation
    (parse-optimization-declaration
     -- Parse.term
     >> register-optimization);

fun exit-phase thy =
  Local-Theory.background-theory (RWList.exit-phase) thy

fun begin-phase name thy =
  Proof-Context.init-global (RWList.enter-phase name thy)

fun
  pretty-rewrite rewrite = Syntax.pretty-term @{context} (rewrite-to-term rewrite)

fun print-optimizations rewrites =
  let
    fun print-rule tact =
      Pretty.block [
        Pretty.str ((#name tact) ^: ),
        pretty-rewrite (#rewrite tact)
      ];
  in
    [Pretty.big-list optimizations: (map print-rule rewrites)]
  end

fun print-phase (phase: phase option) =
  case phase of
    NONE => [Pretty.str no phase] |
    SOME phase =>
  [Pretty.str (phase: ^(#name phase))]
  @ (print-optimizations (#rewrites phase))

fun print-phase-state thy =
  case RWList.get (Proof-Context.theory-of thy) of
    NoPhase - => [Pretty.str not in a phase] |
    InPhase (name, (dom, map)) => print-phase (map name)

fun print-all-phases thy =
  case RWList.get thy of
    NoPhase (dom, store) =>
      let val - = @{print} dom;
      in List.foldr (fn (name, acc) => print-phase (store name) @ acc) [] dom end
```

```
|
   InPhase (name, (dom, store)) => List.foldr (fn (name, acc) => print-phase
(store name) @ acc) [] dom

fun phase-theory-init name thy =
  Local-Theory.init
    {background-naming = Sign.naming-of thy,
        setup = begin-phase name,
        conclude = exit-phase}
    {define = Generic-Target.define Generic-Target.theory-target-foundation,
        notes = Generic-Target.notes Generic-Target.theory-target-notes,
        abbrev = Generic-Target.abbrev Generic-Target.theory-target-abbrev,
        declaration = K Generic-Target.theory-declaration,
        theory-registration = Locale.add-registration-theory,
        locale-dependency = fn - => error Not possible in instantiation target,
        pretty = print-phase-state}
    thy

val - =
  Outer-Syntax.command command-keyword‹phase› instantiate and prove type
arity
   (Parse.name --| Parse.begin
     >> (fn name => Toplevel.begin-main-target true (phase-theory-init name)));


fun apply-print-optimizations thy =
  (print-all-phases thy |> Pretty.writeln-chunks)


val - =
  Outer-Syntax.command command-keyword‹print-optimizations›
    print debug information for optimizations
    (Scan.succeed
      (Toplevel.keep (apply-print-optimizations o Toplevel.theory-of)));
›
```

setup ‹*RWList.reset*›

phase *Canonicalization* **begin**

optimization *constant-add*:
  $(e1 + e2) \mapsto r$ *when* $(e1 = ConstantExpr\ v1 \wedge e2 = ConstantExpr\ v2 \wedge r = ConstantExpr\ (intval\text{-}add\ v1\ v2))$
  unfolding *le-expr-def* **apply** (*cases*; *auto*) **using** *evaltree.ConstantExpr* **defer**
   apply *simp*
  sorry

```
optimization constant-add:
  (c1 + c2) ↦ ConstantExpr (intval-add val-c1 val-c2)
  unfolding le-expr-def apply (cases; auto) using evaltree.ConstantExpr defer
   apply simp
  sorry

print-context
print-optimizations

optimization constant-shift:
  (c + e) ↦ (e + c) when (¬(is-ConstantExpr e) ∧ type e = Integer)
   unfolding rewrite-obligation.simps apply (rule impI) defer apply simp
  sorry

optimization neutral-zero:
  (e + const(0)) ↦ e when (type e = Integer)
   defer apply simp+
  sorry

ML-val ‹@{term (e1 − e2) + e2 ↦ e1}›

optimization neutral-left-add-sub:
  (e1 − e2) + e2 ↦ e1
  sorry

optimization neutral-right-add-sub:
  e1 + (e2 − e1) ↦ e2
  sorry

optimization add-ynegate:
  (x + (−y)) ↦ (x − y) when (type x = Integer ∧ type-safe x y)
  sorry

print-context
print-optimizations



end

print-context
print-optimizations

phase DirectTranslationTest begin

optimization AbsIdempotence: abs(abs(e)) ↦ abs(e) when is-IntegerStamp (stamp-expr
e)
  apply auto
```

```
 by (metis UnaryExpr abs-abs-is-abs stamp-implies-valid-value is-IntegerStamp-def
unary-eval.simps(1))

optimization AbsNegate: abs(−e) ↦ abs(e) when is-IntegerStamp (stamp-expr e)
  apply auto
 by (metis UnaryExpr abs-neg-is-neg stamp-implies-valid-value is-IntegerStamp-def
unary-eval.simps(1))

lemma int-constants-valid:
  assumes is-int-val val
  shows valid-value (constantAsStamp val) val
  using assms apply (cases val)
  by simp+

lemma unary-eval-preserves-validity:
  assumes is-int-val c
  shows valid-value (constantAsStamp (unary-eval op c)) (unary-eval op c)
  using assms apply (cases c) apply simp
     defer defer apply simp+
  apply (cases op)
 using int-constants-valid intval-abs.simps(1) is-int-val.simps(1) unary-eval.simps(1)
apply presburger
 using int-constants-valid intval-negate.simps(1) is-int-val.simps(1) unary-eval.simps(2)
apply presburger
 using int-constants-valid intval-not.simps(1) is-int-val.simps(1) unary-eval.simps(3)
apply presburger
   using int-constants-valid is-int-val.simps(1) unary-eval.simps(4) apply pres-
burger
     defer defer defer
     apply (cases op)
 using int-constants-valid intval-abs.simps(2) is-int-val.simps(2) unary-eval.simps(1)
apply presburger
 using int-constants-valid intval-negate.simps(2) is-int-val.simps(2) unary-eval.simps(2)
apply presburger
 using int-constants-valid intval-not.simps(2) is-int-val.simps(2) unary-eval.simps(3)
apply presburger
  sorry

optimization UnaryConstantFold: UnaryExpr op c ↦ ConstantExpr (unary-eval
op val-c) when is-int-val val-c
  apply (auto simp: int-constants-valid)
  using evaltree.ConstantExpr int-constants-valid unary-eval-preserves-validity by
simp

optimization AndEqual: (x & x) ↦ x when is-IntegerStamp (stamp-expr x)
  apply simp
   apply (metis BinaryExprE CanonicalizeAndProof and-same)
  unfolding size.simps
  by (simp add: size-gt-0)
```

optimization *AndShiftConstantRight*: ((*ConstantExpr x*) + *y*) ↦ *y* + (*ConstantExpr x*) *when* ~(*is-ConstantExpr y*)
  apply *simp*
   apply (*smt* (*verit*, *ccfv-threshold*) *BinaryExprE bin-eval.simps*(*1*) *evaltree.simps intval-add-sym*)
  unfolding *size.simps* **using** *nonconstants-gt-one* **by** *auto*

lemma *neutral-and*:
  assumes *valid-value* (*IntegerStamp 32 lox hix*) *x*
  shows *bin-eval BinAnd x* (*IntVal32* (−*1*)) = *x*
  using *assms bin-eval.simps*(*4*) **by** (*cases x*; *auto*)

optimization *AndNeutral*: (*x* & (*const* (*NOT 0*))) ↦ *x when* (*stamp-expr x* = *IntegerStamp 32 l u*)
   apply *simp*
  using *neutral-and stamp-implies-valid-value* **apply** *auto*
  by *metis*

optimization *ConditionalEqualBranches*: (*b ? v* : *v*) ↦ *v*
  apply *simp*
   apply *force*
  unfolding *size.simps*
  by *auto*

optimization *ConditionalEqualIsRHS*: ((*x* == *y*) *? x* : *y*) ↦ *y when* (*type x* = *Integer* ∧ *type-safe x y*)
   apply *simp*
  apply (*smt* (*verit*, *del-insts*) *BinaryExprE CanonicalizeConditionalProof ConditionalExprE cond-eq type-safe-def unfold-type*)
  unfolding *size.simps* **by** *simp*

lemma *bool-is-int-val*:
  is-int-val (*bool-to-val x*)
  using *bool-to-val.simps is-int-val.simps* **by** (*metis* (*full-types*))

lemma *bin-eval-preserves-validity*:
  assumes *int-and-equal-bits c1 c2*
  shows *valid-value* (*constantAsStamp* (*bin-eval op c1 c2*)) (*bin-eval op c1 c2*)
  using *assms* **apply** (*cases c1*; *cases c2*; *auto*)
    apply (*cases op*; *auto*)
  using *int-constants-valid bool-is-int-val*
 apply (*metis* (*full-types*) *IRTreeEval.bool-to-val.simps*(*1*) *IRTreeEval.bool-to-val.simps*(*2*) *Values.bool-to-val.simps*(*1*) *Values.bool-to-val.simps*(*2*))
  using *int-constants-valid bool-is-int-val*
 apply (*metis* (*full-types*) *IRTreeEval.bool-to-val.simps*(*1*) *IRTreeEval.bool-to-val.simps*(*2*)

Values.bool-to-val.simps(1) Values.bool-to-val.simps(2))
  using *int-constants-valid bool-is-int-val*
 apply (*metis* (*full-types*) *IRTreeEval.bool-to-val.simps(1) IRTreeEval.bool-to-val.simps(2) Values.bool-to-val.simps(1) Values.bool-to-val.simps(2)*)
   apply (*cases op*; *auto*)
  using *int-constants-valid bool-is-int-val*
 apply (*metis* (*full-types*) *IRTreeEval.bool-to-val.simps(1) IRTreeEval.bool-to-val.simps(2) Values.bool-to-val.simps(1) Values.bool-to-val.simps(2)*)
  using *int-constants-valid bool-is-int-val*
 apply (*metis* (*full-types*) *IRTreeEval.bool-to-val.simps(1) IRTreeEval.bool-to-val.simps(2) Values.bool-to-val.simps(1) Values.bool-to-val.simps(2)*)
  using *int-constants-valid bool-is-int-val*
 by (*metis* (*full-types*) *IRTreeEval.bool-to-val.simps(1) IRTreeEval.bool-to-val.simps(2) Values.bool-to-val.simps(1) Values.bool-to-val.simps(2)*)

optimization *BinaryFoldConstant*: *BinaryExpr op* (*ConstantExpr e1*) (*ConstantExpr e2*) $\mapsto$ *ConstantExpr* (*bin-eval op e1 e2*) *when int-and-equal-bits e1 e2*
  apply *simp* **using** *evaltree.BinaryExpr evaltree.ConstantExpr stamp-implies-valid-value*
  using *bin-eval-preserves-validity*
  apply *force* **using** *nonconstants-gt-one*
  by *auto*

optimization *AddShiftConstantRight*: ((*ConstantExpr x*) + *y*) $\mapsto$ *y* + (*ConstantExpr x*) *when* ~(*is-ConstantExpr y*)
  apply *simp*
   apply (*smt* (*verit*, *del-insts*) *BinaryExprE bin-eval.simps(1) evaltree.simps int-val-add-sym*)
  unfolding *size.simps* **using** *nonconstants-gt-one* **by** *simp*

lemma *neutral-add*:
  assumes *valid-value* (*IntegerStamp 32 lox hix*) *x*
  shows *bin-eval BinAdd x* (*IntVal32* (*0*)) = *x*
  using *assms bin-eval.simps(4)* **by** (*cases x*; *auto*)

optimization *AddNeutral*: (*e* + (*const 0*)) $\mapsto$ *e when* (*stamp-expr e* = *IntegerStamp 32 l u*)
   apply *simp* **using** *neutral-add stamp-implies-valid-value*
  using *evaltree.BinaryExpr evaltree.ConstantExpr*
   apply (*metis* (*no-types*, *hide-lams*) *BinaryExprE ConstantExprE*)
  unfolding *size.simps* **by** *simp*

lemma *intval-negateadd-equals-sub-left*: *bin-eval BinAdd* (*unary-eval UnaryNeg e*) *y* = *bin-eval BinSub y e*
  by (*cases e*; *auto*; *cases y*; *auto*)

lemma *intval-negateadd-equals-sub-right*: *bin-eval BinAdd x* (*unary-eval UnaryNeg e*) = *bin-eval BinSub x e*
  by (*cases e*; *auto*; *cases x*; *auto*)

optimization *AddLeftNegateToSub*: $-e + y \mapsto y - e$
apply *simp* **using** *intval-negateadd-equals-sub-left*
apply (*metis BinaryExpr BinaryExprE UnaryExprE*)
unfolding *size.simps*
by *simp*

optimization *AddRightNegateToSub*: $x + -e \mapsto x - e$
apply *simp* **using** *intval-negateadd-equals-sub-right*
apply (*metis BinaryExpr BinaryExprE UnaryExprE*)
unfolding *size.simps*
by *simp*

optimization *AddShiftConstantRight*: ((*ConstantExpr x*) + *y*) $\mapsto$ *y* + (*ConstantExpr x*) *when* ~ (*is-ConstantExpr y*)
apply *simp*
apply (*metis BinaryExpr BinaryExprE bin-eval.simps*(*1*) *intval-add-sym*)
unfolding *size.simps* **using** *nonconstants-gt-one* **by** *simp*

optimization *MulEliminator*: (*x* $*$ *const*(*0*)) $\mapsto$ *const*(*0*) *when* (*stamp-expr x* = *IntegerStamp 32 l u*)
apply *simp*
apply (*metis BinaryExprE ConstantExprE annihilator-rewrite-helper*(*1*) *bin-eval.simps*(*2*) *stamp-implies-valid-value*)
unfolding *size.simps*
by (*simp add*: *size-gt-0*)

optimization *MulNeutral*: (*x* $*$ *const*(*1*)) $\mapsto$ *x when* (*stamp-expr x* = *IntegerStamp 32 l u*)
apply *simp*
apply (*metis BinaryExprE ConstantExprE bin-eval.simps*(*2*) *neutral-rewrite-helper*(*1*) *stamp-implies-valid-value*)
unfolding *size.simps* **by** *simp*

value (*3*::*32 word*) *mod 32*

lemma (*x*::*nat*) $\geq$ *0* $\wedge$ *x* < *base* $\Longrightarrow$ *x mod base* = *x*
sledgehammer
using *mod-less* **by** *blast*

lemma *word-mod-less*: (*x*::('*a*::*len*) *word*) < *base* $\Longrightarrow$ *x mod base* = *x*
by (*metis mod-less not-le unat-arith-simps*(*2*) *unat-arith-simps*(*7*) *unat-mono word-le-less-eq*)

value *4294967298*::*32 word*

lemma *shift-equality*: ((*v1*::*32 word*) << *unat* ((*v2*::*32 word*) *mod 32*)) = *v1* $*$ ((*2*

```
^ (unat v2))::32 word)
proof –
  have size-class.size (2 ^ (unat v2)) = 32 sorry
  then have (2 ^ (unat v2)) < 2 ^ 32
    using uint-range-size sorry
  then have unat v2 < 32
    using nat-power-less-imp-less zero-less-numeral by blast
  then show ?thesis
    using ‹2 ^ unat v2 < 2 ^ 32› numeral-Bit0 power2-eq-square power-add sorry
qed


print-context
print-optimizations
end

print-optimizations

end
```