# Veriopt Theories

February 9, 2022

## Contents

## 1 Canonicalization Phase

```
theory Common
  imports
    OptimizationDSL.Canonicalization
    HOL−Eisbach.Eisbach
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 c) = 1 |
  size (ParameterExpr ind s) = 2 |
  size (LeafExpr nid s) = 2 |
  size (ConstantVar c) = 2 |
  size (VariableExpr x s) = 2

method unfold-optimization =
  (unfold rewrite-preservation.simps, unfold rewrite-termination.simps,
    unfold intval.simps,
    rule conjE, simp, simp del: le-expr-def)
  | (unfold rewrite-preservation.simps, unfold rewrite-termination.simps,
    rule conjE, simp, simp del: le-expr-def)

end
```

### 1.1 Conditional Expression

```
theory ConditionalPhase
  imports
```

```
    Common
    Proofs.StampEvalThms
begin

phase Conditional
  terminating size
begin

lemma negates: is-IntVal32 e ∨ is-IntVal64 e ⟹ val-to-bool (val[e]) ≡ ¬(val-to-bool
(val[¬e]))
   by (smt (verit, best) Value.disc(1) Value.disc(10) Value.disc(4) Value.disc(5)
Value.disc(6) Value.disc(9) intval-logic-negation.elims val-to-bool.simps(1) val-to-bool.simps(2)
zero-neq-one)

optimization negate-condition: ((¬e) ? x : y) ⟼ (e ? y : x)
    apply unfold-optimization apply simp using negates
   using ConditionalExprE UnaryExprE intval-logic-negation.elims unary-eval.simps(4)
val-to-bool.simps(1) val-to-bool.simps(2) zero-neq-one
    apply (smt (verit) ConditionalExpr)
    unfolding size.simps by simp

optimization const-true: (true ? x : y) ⟼ x
   apply unfold-optimization
   apply force
  unfolding size.simps by simp

optimization const-false: (false ? x : y) ⟼ y
   apply unfold-optimization
   apply force
  unfolding size.simps by simp

optimization equal-branches: (e ? x : x) ⟼ x
   apply unfold-optimization
   apply force
  unfolding size.simps by auto


definition wff-stamps :: bool where
  wff-stamps = (∀ m p expr val . ([m,p] ⊢ expr ↦ val) ⟶ valid-value val (stamp-expr
expr))

definition wf-stamp :: IRExpr ⇒ bool where
  wf-stamp e = (∀ m p v. ([m, p] ⊢ e ↦ v) ⟶ valid-value v (stamp-expr e))

optimization condition-bounds-x: ((u < v) ? x : y) ⟼ x
    when (stamp-under (stamp-expr u) (stamp-expr v) ∧ wf-stamp u ∧ wf-stamp v)
   apply unfold-optimization
  using stamp-under-semantics
  using wf-stamp-def
```

**apply** (*smt* (*verit*, *best*) *ConditionalExprE le-expr-def stamp-under.simps*)
**unfolding** *size.simps* **by** *simp*

**optimization** *condition-bounds-y*: $((x < y)\ ?\ x : y) \longmapsto y$
*when* (*stamp-under* (*stamp-expr y*) (*stamp-expr x*) ∧ *wf-stamp x* ∧ *wf-stamp y*)
**apply** *unfold-optimization*
**using** *stamp-under-semantics-inversed*
**using** *wf-stamp-def*
**apply** (*smt* (*verit*, *best*) *ConditionalExprE le-expr-def stamp-under.simps*)
**unfolding** *size.simps* **by** *simp*

**optimization** *b*[*intval*]: ((*x eq y*) *?* *x* : *y*) ⟼ *y*
**apply** *unfold-optimization*
**apply** (*smt* (*z3*) *bool-to-val.simps*(*2*) *intval-equals.elims val-to-bool.simps*(*1*) *val-to-bool.simps*(*3*))
**unfolding** *intval.simps*
**apply** (*smt* (*z3*) *BinaryExprE ConditionalExprE Value.inject*(*1*) *Value.inject*(*2*) *bin-eval.simps*(*10*) *bool-to-val.simps*(*2*) *evalDet intval-equals.simps*(*1*) *intval-equals.simps*(*10*) *intval-equals.simps*(*12*) *intval-equals.simps*(*15*) *intval-equals.simps*(*16*) *intval-equals.simps*(*2*) *intval-equals.simps*(*5*) *intval-equals.simps*(*8*) *intval-equals.simps*(*9*) *le-expr-def val-to-bool.cases val-to-bool.elims*(*2*))
**unfolding** *size.simps* **by** *auto*

**end**

**end**