We present here a mechanically verified Isabelle/HOL proof that a program assertion will have the same truth value whether it is interpreted in a classical 2-valued logic or 3-valued logic provided it is defined—in accordance with the definedness predicate transformer detailed in the paper. In fact, we prove both the soundness and completeness of our approach. Note that this appendix is automatically generated from the Isabelle/HOL theory files.

As can be expected, the proof is syntax directed. We start by defining our expression language syntax.

### A Expression Syntax

**theory Syntax imports Main begin**

Our logic language is inspired by C expressions in that it consists of integer expressions which can also be interpreted as boolean expressions by following the C convention of 0 as false and anything else as true.

**types** 

- Val = int — universe of values

Identifiers, used to name program variables, function and predicate symbols, are simply represented by an index into a symbol set.

**types** 

- Ident = nat

Though the expression language is minimal, it can be used to embed first order predicate logic with function symbols, predicate symbols and equality.

**datatype** Expr  

- = Undefined — an expression that is always undefined
- = Const Val — constants
- = Var Ident — variables
- = IfExpr Expr Expr Expr — if-then-else
- = AllExpr Ident Expr — universal quantifier

Without loss of generality, we model functions and predicates as having a single argument. Note how the if-then-else term constructor is named IfExpr and then given the abbreviation If rather than directly naming the constructor If. This has been done because If has a special internal meaning for Isabelle which causes problems if the type constructor is directly named If but it is safe to use as an abbreviation. This idiom is repeated here for AllExpr and elsewhere—e.g. falseVal and falseExpr below, even overloading abbreviations at times.

While we do not have a single representation for ‘true’, we sometimes return 1 when no other value is readily available.

**constdefs**

- falseVal :: Val (false) falseVal == 0
- trueVal1 :: Val (true1) trueVal1 == 1

We define syntactic sugars for true and false (as expressions) and some of the propositional connectives.

**constdefs**

- falseExpr :: Expr (false)

Using the if-then-else operator we define conditional conjunction and strict conjunction along with abbreviations using the familiar Java syntax & & and &., respectively.

**constdefs**

- cconj :: Expr ⇒ Expr ⇒ Expr (infix & & 60) — conditional conjunction
- conj :: Expr ⇒ Expr ⇒ Expr (infix 60) — strict conjunction

For strict conjunction, both operands need to be defined for the overall expression to be defined. Hence, even if the first operand is false, we must still evaluate the second operand.

**end**

### B Semantic Domains

**theory SemanticDomain imports Syntax begin**

#### B.1 Lifted Value Domain

In a three-valued logic, an expression can either evaluate to a value or be undefined. In Isabelle, one typically models this as an option type.

**syntax**

- option :: type => type (⊥ [1000] 1000)

**notation**

- None (⊥) and Some ([|]) and the ([|])

Thus, in Val⊥, ⊥ represents the undefined value; given a value v, its lifted form is represented by [v]. The inverse of lift is represented as [ ].

In our expression language, multiple values represent true; the following predicate tests for truth over Val⊥.

**fun** 

- isTrue :: Val⊥ ⇒ bool where
  - isTrue ⊥ = False | isTrue ([v]) = (v ≠ false)

**lemma** isTrue x = (x ≠ ⊥ ∧ x ≠ [false]) by (case-tac x, auto)

#### B.2 Denotations for Functions

Total functions in our language map Vals into Vals. Functions in Isabelle are total. We follow the common idiom of modeling partial functions by making the range an option type.

**types**

- TFuncDen = Val ⇒ Val — total functions
- PFuncDen = Val ⇒ Val⊥ — partial functions

In general in our expression language, a function symbol represents a partial function. Given a function identifier, the following map returns the function’s denotation.
const
\text{pFunVal} :: \text{Ident} \Rightarrow \text{PFunDen} ([\cdot])

The domain of a partial function is the set of values over which it is defined.
\text{fun} \ pFunDom :: \text{Ident} \Rightarrow \text{Val set (dom) where}
\text{dom } f = \{ v : ([f] v) \neq \perp \}

We can “totalize” a given partial function by mapping elements outside its domain to arbitrary values.
\text{fun} \ IFunVal :: \text{Ident} \Rightarrow \text{IFunDen ([\cdot] \text{total}) where}
\text{[f] \text{total} v = (if } v \in \text{dom } f \text{ then } ([f] v) \text{ else arbitrary)}

It follows from the definition that a function matches its totalized version for values over its domain.
\text{lemma} \ v \in \text{dom } f \implies ([f] \text{total} v = ([f] v) \text{ by (simp)}

B.3 Program State

We model program state (used to hold the values of program variables that occur in expressions) as a simple mapping from identifiers to their values.
\text{types}
\text{State = Ident} \Rightarrow \text{Val}

B.4 Syntactic sugar

We define an Isabelle syntactic shorthand that will allow us to make the definitions of our semantic functions more readable. It is an approximation of a monadic bind operator:

\text{syntax}
\text{-bind :: patterns} \Rightarrow \text{Val option} \Rightarrow \text{Val} \Rightarrow \text{Val} \quad ((\vdash \vdash; \vdash; \vdash) 0)
\text{translations}
\text{v := E; } E' \implies (\text{let } v = E \text{ if } v = \perp \text{ then } \perp \text{ else } E')

In an earlier version of the theory we used syntax based on a real monadic bind, but unfortunately this rendered the proofs more complex.

C CLASSICAL SEMANTICS

theory Semantics3 imports Syntax SemanticDomain begin

In this section we present an interpretation of our expression language as formulae in classical two-valued logic.

C.1 Main Semantic Function

Since expressions contain program variables, their interpretation must be done relative to a state. Note that there is no case for \text{Undef}, hence its value is left unspecified.
\text{fun} \ Eval :: \text{Expr} \Rightarrow \text{State} \Rightarrow \text{Val} (\text{E}_2 [\cdot] \cdot) \text{ where}
\text{E}_2 [\text{Const } c]_s = c
\text{E}_2 [\text{Var } i]_s = s i
\text{E}_2 [\text{Fun } f] e]_s = [f] \text{total}(\text{E}_2 [e]_s)
\text{E}_2 [\text{if } c \text{ t} e]_s = (\text{if } (\text{E}_2 [c]_s) = \text{false then } \text{E}_2 [e]_s \text{ else } \text{E}_2 [t]_s)
\text{E}_2 [\text{All } i e]_s = (\text{if } (\exists v. \text{E}_2 [e]_s(i:=v) = \perp) \text{ then } \perp \text{ else if } (\exists v. \text{E}_2 [e]_s(i:=v) = [false]) \text{ then } \text{false else true1})

C.2 An Encoding of Truth Tables

Basic sanity “tests”: essentially representing the truth tables for each propositional connective.
\text{lemma} \ \text{E}_2 [\text{false}]_s = \text{false} \land \text{E}_2 [\text{true}]_s = \text{true1} \text{ by auto}

There is not much that can be said in classical two valued logic about \text{UnDef} other than it has some value.
\text{lemma} \ \text{E}_2 [\text{UnDef}]_s = \text{E}_2 [\text{UnDef}]_s \text{ by auto}

\text{lemma} \ \text{E}_2 [\text{not false}]_s = \text{false} \land \text{E}_2 [\text{not true}]_s = \text{true1} \text{ by auto}

\text{lemma} \ \text{E}_2 [\text{false } \land \text{false}]_s = \text{false} \land \text{E}_2 [\text{true } \land \text{true}]_s = \text{true1}
\text{lemma} \ \text{E}_2 [\text{false } \land \text{true}]_s = \text{true1}
\text{lemma} \ \text{E}_2 [\text{true } \land \text{false}]_s = \text{false} \land \text{E}_2 [\text{true } \land \text{true}]_s = \text{true1}
\text{lemma} \ \text{E}_2 [\text{false } \land \text{true}]_s = \text{false}
\text{lemma} \ \text{E}_2 [\text{true } \land \text{false}]_s = \text{false} \land \text{E}_2 [\text{true } \land \text{false}]_s = \text{true1}
\text{lemma} \ \text{E}_2 [\text{true } \land \text{false}]_s = \text{false}
\text{lemma} \ \text{E}_2 [\text{true } \land \text{false}]_s = \text{false} \land \text{E}_2 [\text{true } \land \text{false}]_s = \text{true1}
\text{lemma} \ \text{E}_2 [\text{true } \land \text{false}]_s = \text{false}

D SEMANTICS FOR A THREE-VALUED LOGIC

theory Semantics3imports Syntax SemanticDomain begin

D.1 Main Semantic Function

We now define the interpretation of expressions as formulae in a three-valued logic:
\text{fun} \ Eval :: \text{Expr} \Rightarrow \text{State} \Rightarrow \text{Val}_3 (\text{E}_3 [\cdot] \cdot) \text{ where}
\text{E}_3 [\text{UnDef}]_s = \perp
\text{E}_3 [\text{Const } c]_s = [c]
\text{E}_3 [\text{Var } i]_s = [s i]
\text{E}_3 [\text{Fun } f] e]_s = (v := \text{E}_3 [e]_s ; [f] [v])
\text{E}_3 [\text{if } c \text{ t} e]_s = (v := \text{E}_3 [e]_s ;
\text{if } v = [false] \text{ then } \text{E}_3 [e]_s \text{ else } \text{E}_3 [t]_s)
\text{E}_3 [\text{All } i e]_s = (\text{if } (\exists v. \text{E}_3 [e]_s(i:=v) = \perp) \text{ then } \perp \text{ else if } (\exists v. \text{E}_3 [e]_s(i:=v) = [false]) \text{ then } \text{false else true1})

D.2 An Encoding of Truth Tables

Truth tables for the propositional connectives as interpreted in three-valued logic:
\text{declare Let-def(simp)

\text{lemma} \ \text{E}_3 [\text{false}]_s = [false]
\text{lemma} \ \text{E}_3 [\text{true}]_s = [true1]
\text{lemma} \ \text{E}_3 [\text{UnDef}]_s = \perp \text{ by auto}
\text{lemma} \ \text{E}_3 [\text{true }]_s = [true1]
\text{lemma} \ \text{E}_3 [\text{false}]_s = \perp \text{ by auto}
\text{lemma} \ \text{E}_3 [\text{false } \land \text{false}]_s = [false]
\text{lemma} \ \text{E}_3 [\text{false } \land \text{true}]_s = [true1]
\text{lemma} \ \text{E}_3 [\text{true } \land \text{false}]_s = \perp \text{ by auto}
\text{lemma} \ \text{E}_3 [\text{false } \land \text{true}]_s = [false]
\text{lemma} \ \text{E}_3 [\text{true } \land \text{false}]_s = \perp \text{ by auto}
\text{lemma} \ \text{E}_3 [\text{false } \land \text{true}]_s = [false]
∧ \mathcal{E}_3[true \land true]_s = [true1]
∧ \mathcal{E}_3[true \land false]_s = [false]
∧ \mathcal{E}_3[true \land Undefined]_s = ⊥ by simp

lemma \mathcal{E}_3[true \land true]_s = [true1]
∧ \mathcal{E}_3[true \land false]_s = [false]
∧ \mathcal{E}_3[true \land Undefined]_s = ⊥
∧ \mathcal{E}_3[false \land true]_s = [false]
∧ \mathcal{E}_3[false \land false]_s = [false]
∧ \mathcal{E}_3[false \land Undefined]_s = ⊥ by simp

Conjunction (op \land) is strict:
lemma \mathcal{E}_3[Undefined \land e]_s = ⊥
∧ \mathcal{E}_3[e \land Undefined]_s = ⊥ by auto

D.3 The Definedness Predicate

The definedness predicate, as expressed in three-valued logic, holds when a given expression evaluates to a defined value:

fun \mathcal{D} :: State ⇒ Expr ⇒ bool (\mathcal{D} -)
where
\mathcal{D}s e = (\mathcal{E}_3[e]_s \neq ⊥)

The following predicate holds when an expression is defined in all states.

fun \mathcal{isDefAll} :: Expr ⇒ bool (\mathcal{D} ∀)
where
\mathcal{D} ∀e = (∀ s . \mathcal{D}s e)

As expected, the undefined expression is never defined, constants and variables are always defined:

lemma ∼ \mathcal{D} ∀ Undefined ∧ \mathcal{D} ∀ (Const c) ∧ \mathcal{D} ∀ (Var i) by auto

Some basic results about quantifiers:

lemma \mathcal{E}_3[\forall i Undefined]_s = ⊥
∧ \mathcal{E}_3[\forall i true]_s = [true1]
∧ \mathcal{E}_3[\forall i false]_s = [false] by auto

end

E  EQUIVALENCE

theory Equiv imports Semantics2 Semantics3 begin

This theory presents our main theorem, namely, that if an expression is defined, then interpreting it in a 2-valued or 3-valued logic yields the same result.

theorem soundness: ∀ s . \mathcal{D}s e → \mathcal{E}_3[e]_s = [\mathcal{E}_2[e]_s]
proof (induct e)
  case Undefined show ?case by simp
next case Const show ?case by simp
next case Var show ?case by simp
next case (Fun i e) thus ?case by auto
next case (HExpr c t e) thus ?case by auto
next case (AllExpr i e) thus \forall i e.
  ∀ s . \mathcal{D}s e → \mathcal{E}_3[e]_s = [\mathcal{E}_2[e]_s] \implies
  ∀ s . \mathcal{D}s (All i e) → \mathcal{E}_3[All i e]_s = [\mathcal{E}_2[All i e]_s]
  by (auto, rule-tac x=v in exI, auto)
qed

In fact, if the 2-valued and 3-valued interpretations of a formula coincide, then the formula must be defined.

theorem completeness: ∀ s . \mathcal{E}_3[e]_s = [\mathcal{E}_2[e]_s] → \mathcal{D}s e
by (induct e, auto)
end