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*
  = *Undef* — an expression that is always undefined
  | *Const Val* — constants
  | *Var Ident* — variables
  | *Fun Ident Expr* — strict functions and predicates
  | *IfExpr Expr Expr Expr* (*If*) — non-strict if-then-else
  | *AllExpr Ident Expr* (*All*) — 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 syntatic sugars for true and false (as expressions) and some of the propositional connectives.

constdefs
  falseExpr :: *Expr* (*false*)
    falseExpr == *Const falseVal*
  true1Expr :: *Expr* (*true1*)
    true1 == *Const trueVal1*
  not :: *Expr* ⇒ *Expr*
    not e == *If e false true1*
  trueExpr :: *Expr* (*true*)
    trueExpr == *not 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
    e1 && *e2* == *If e1 e2 false*
  conj :: *Expr* ⇒ *Expr* ⇒ *Expr* (**infix** & *60*) — strict conjunction
    e1 & *e2* == *If e1 e2* (*If e2 false false*)

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* ($-_\perp$ [*1000*] *1000*)
notation
  None ($\perp$) **and** *Some* ($\lfloor - \rfloor$) **and** *the* ($\lceil - \rceil$)

Thus, in $Val_\perp$, $\perp$ represents the undefined value; given a value v, its lifted form is represented by $\lfloor v \rfloor$. The inverse of lift is represented as $\lceil - \rceil$.

In our expression language, multiple values represent *true*; the following predicate tests for truth over $Val_\perp$.

fun *isTrue* :: $Val_\perp \Rightarrow$ *bool* **where**
  $isTrue\ \perp = False \mid isTrue\ (\lfloor v \rfloor) = (v \neq false)$

lemma $isTrue\ x = (x \neq \perp \wedge x \neq \lfloor false \rfloor)$ **by** (*case-tac x*, *auto*)

B.2 Denotations for Functions

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

types
  TFunDen = *Val* ⇒ *Val* — total functions
  PFunDen = $Val \Rightarrow Val_\perp$ — 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.

consts
$pFunVal :: Ident \Rightarrow PFunDen\ ([\![\text{-}]\!])$

The domain of a partial function is the set of values over which it is defined.

fun $pFunDom :: Ident \Rightarrow Val\ set\ (dom)$ **where**
$dom\ f = \{\ v\ .\ ([\![f]\!]\ v) \neq \bot\}$

We can "totalize" a given partial function by mapping elements outside its domain to arbitrary values.

fun $tFunVal :: Ident \Rightarrow TFunDen\ ([\![\text{-}]\!]^{total})$ **where**
$[\![f]\!]^{total}\ v = (if\ v \in dom\ f\ then\ \lceil [\![f]\!]\ v \rceil\ \ else\ arbitrary)$

It follows from the definition that a function matches its totalized version for values over its domain.

lemma $v \in dom\ f \Longrightarrow [\![f]\!]^{total}\ v = \lceil [\![f]\!]\ v \rceil$ **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.

types
$State = Ident \Rightarrow Val$

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:

syntax
$\text{-}bind :: patterns \Rightarrow Val\ option \Rightarrow Val \Rightarrow Val \qquad ((\text{-} := \text{-};;/\text{/-})\ 0)$
translations
$v := E;;\ E' \Rightarrow (let\ v = E\ in\ if\ v = \bot\ then\ \bot\ else\ E')$

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

end

C Classical Semantics

theory *Semantics2* **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 *Undef*, hence its value is left unspecified.

fun $Eval2 :: Expr \Rightarrow State \Rightarrow Val\ (\mathcal{E}_2[\![\text{-}]\!]\text{-})$ **where**
$\mathcal{E}_2[\![Const\ c]\!]_s = c$
$|\ \mathcal{E}_2[\![Var\ i]\!]_s = s\ i$
$|\ \mathcal{E}_2[\![Fun\ f\ e]\!]_s = [\![f]\!]^{total}(\mathcal{E}_2[\![e]\!]_s)$
$|\ \mathcal{E}_2[\![If\ c\ t\ e]\!]_s = (if\ (\mathcal{E}_2[\![c]\!]_s) = false\ then\ \mathcal{E}_2[\![e]\!]_s\ else\ \mathcal{E}_2[\![t]\!]_s)$
$|\ \mathcal{E}_2[\![All\ i\ e]\!]_s = (if\ (\exists\ v.\ \mathcal{E}_2[\![e]\!]_{s(i:=v)} = false)$
$\quad then\ false\ else\ true1)$

C.2 An Encoding of Truth Tables

Basic sanity "tests": essentially representing the truth tables for each propositional connective.

lemma $\mathcal{E}_2[\![false]\!]_s = false \wedge \mathcal{E}_2[\![true]\!]_s\ = true1$ **by** *auto*

There is not much that can be said in classical two valued logic about *Undef* other than it has some value.

lemma $\mathcal{E}_2[\![Undef]\!]_s = \mathcal{E}_2[\![Undef]\!]_s$ **by** *auto*

lemma $\mathcal{E}_2[\![not\ true]\!]_s\ = false$
$\wedge\ \mathcal{E}_2[\![not\ false]\!]_s = true1$ **by** *auto*

lemma $\mathcal{E}_2[\![false\ \&\&\ e]\!]_s = false$
$\wedge\ \mathcal{E}_2[\![true\ \&\&\ true]\!]_s\ = true1$
$\wedge\ \mathcal{E}_2[\![true\ \&\&\ false]\!]_s = false$ **by** *simp*

lemma $\mathcal{E}_2[\![true\ \&\ \ true]\!]_s = true1$
$\wedge\ \mathcal{E}_2[\![true\ \&\ false]\!]_s = false$
$\wedge\ \mathcal{E}_2[\![false\ \&\ true]\!]_s\ = false$
$\wedge\ \mathcal{E}_2[\![false\ \&\ false]\!]_s = false$ **by** *simp*

lemma $\mathcal{E}_2[\![false\ \&\ e]\!]_s = false$
$\wedge\ \mathcal{E}_2[\![e\ \&\ false]\!]_s = false$ **by** *simp*

lemma $\mathcal{E}_2[\![All\ i\ true]\!]_s = true1 \wedge \mathcal{E}_2[\![All\ i\ false]\!]_s = false$ **by** *auto*

end

D Semantics for a Three-valued Logic

theory *Semantics3* **imports** *Syntax SemanticDomain* **begin**

D.1 Main Semantic Function

We now define the interpretation of expressions as formulae in a three-valued logic:

fun $Eval :: Expr \Rightarrow State \Rightarrow Val_\bot\ (\mathcal{E}_3[\![\text{-}]\!]\text{-})$ **where**
$\mathcal{E}_3[\![Undef]\!]_s = \bot$
$|\ \mathcal{E}_3[\![Const\ c]\!]_s = \lfloor c \rfloor$
$|\ \mathcal{E}_3[\![Var\ i]\!]_s = \lfloor s\ i \rfloor$
$|\ \mathcal{E}_3[\![Fun\ f\ e]\!]_s = (v := \mathcal{E}_3[\![e]\!]_s;;\ [\![f]\!]\lceil v \rceil)$
$|\ \mathcal{E}_3[\![If\ c\ t\ e]\!]_s = (v := \mathcal{E}_3[\![c]\!]_s;;$
$\quad if\ v = \lfloor false \rfloor\ then\ \mathcal{E}_3[\![e]\!]_s\ else\ \mathcal{E}_3[\![t]\!]_s)$
$|\ \mathcal{E}_3[\![All\ i\ e]\!]_s = (if\ (\exists\ v\ .\ \mathcal{E}_3[\![e]\!]_{s(i:=v)} = \bot)\ then\ \bot$
$\quad else\ if\ (\exists\ v.\ \mathcal{E}_3[\![e]\!]_{s(i:=v)} = \lfloor false \rfloor)\ then\ \lfloor false \rfloor$
$\quad else\ \lfloor true1 \rfloor)$

D.2 An Encoding of Truth Tables

Truth tables for the propositional connectives as interpreted in three-valued logic:

declare *Let-def* [*simp*]

lemma $\mathcal{E}_3[\![false]\!]_s = \lfloor false \rfloor$
$\wedge\ \mathcal{E}_3[\![true]\!]_s\ = \lfloor true1 \rfloor$
$\wedge\ \mathcal{E}_3[\![Undef]\!]_s\ = \bot$ **by** *auto*

lemma $\mathcal{E}_3[\![not\ true]\!]_s\ = \lfloor false \rfloor$
$\wedge\ \mathcal{E}_3[\![not\ false]\!]_s = \lfloor true1 \rfloor$
$\wedge\ \mathcal{E}_3[\![not\ Undef]\!]_s = \bot$ **by** *auto*

lemma $\mathcal{E}_3[\![Undef\ \&\&\ e]\!]_s = \bot$
$\wedge\ \mathcal{E}_3[\![false\ \&\&\ \ e]\!]_s = \lfloor false \rfloor$

$\wedge\ \mathcal{E}_3[\![true\ \&\&\ true]\!]_s\ = \lfloor true1 \rfloor$
$\wedge\ \mathcal{E}_3[\![true\ \&\&\ false]\!]_s = \lfloor false \rfloor$
$\wedge\ \mathcal{E}_3[\![true\ \&\&\ Undef]\!]_s = \bot$ **by** *simp*

lemma $\mathcal{E}_3[\![true\ \&\ \ true]\!]_s = \lfloor true1 \rfloor$
$\wedge\ \mathcal{E}_3[\![true\ \&\ false]\!]_s = \lfloor false \rfloor$
$\wedge\ \mathcal{E}_3[\![true\ \&\ Undef]\!]_s = \bot$
$\wedge\ \mathcal{E}_3[\![false\ \&\ true]\!]_s\ = \lfloor false \rfloor$
$\wedge\ \mathcal{E}_3[\![false\ \&\ false]\!]_s = \lfloor false \rfloor$
$\wedge\ \mathcal{E}_3[\![false\ \&\ Undef]\!]_s = \bot$ **by** *simp*

Conjunction (*op* &) is strict:

lemma $\mathcal{E}_3[\![Undef\ \&\ e]\!]_s = \bot$
$\wedge\ \mathcal{E}_3[\![e\ \&\ Undef]\!]_s = \bot$ **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* $\Rightarrow$ *Expr* $\Rightarrow$ *bool* ($\mathcal{D}$- -) **where**
$\mathcal{D}_s\ e = (\mathcal{E}_3[\![e]\!]_s \neq \bot)$

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

fun *isDefAll* :: *Expr* $\Rightarrow$ *bool* ($\mathcal{D}_\forall$ -) **where**
$\mathcal{D}_\forall\ e = (\forall\ s\ .\ \mathcal{D}_s\ e)$

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

lemma $\neg\ \mathcal{D}_\forall\, Undef\ \wedge\ \mathcal{D}_\forall\,(Const\ c)\ \wedge\ \mathcal{D}_\forall\,(Var\ i)$ **by** *auto*

Some basic results about quantifiers:

lemma $\mathcal{E}_3[\![All\ i\ Undef]\!]_s = \bot$
$\wedge\ \mathcal{E}_3[\![All\ i\ true]\!]_s = \lfloor true1 \rfloor$
$\wedge\ \mathcal{E}_3[\![All\ i\ false]\!]_s = \lfloor false \rfloor$ **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*: $\forall\ s\ .\ \mathcal{D}_s\ e \longrightarrow \mathcal{E}_3[\![e]\!]_s = \lfloor \mathcal{E}_2[\![e]\!]_s \rfloor$
proof (*induct e*)
  case *Undef* **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 (*IfExpr c t e*) **thus** *?case* **by** *auto*
next case (*AllExpr i e*) **thus** $\bigwedge i\ e.$
$\forall s.\ \mathcal{D}_s\ e \longrightarrow \mathcal{E}_3[\![e]\!]_s = \lfloor \mathcal{E}_2[\![e]\!]_s \rfloor \Longrightarrow$
$\forall s.\ \mathcal{D}_s\ (All\ i\ e) \longrightarrow \mathcal{E}_3[\![All\ i\ e]\!]_s = \lfloor \mathcal{E}_2[\![All\ i\ e]\!]_s \rfloor$
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*: $\forall\ s\ .\ \mathcal{E}_3[\![e]\!]_s = \lfloor \mathcal{E}_2[\![e]\!]_s \rfloor \longrightarrow \mathcal{D}_s\ e$
by (*induct e*, *auto*)
end