Abstract Interpretation

Ranjit Jhala, UC San Diego

April 22, 2013

Fundamental Challenge of Program Analysis

How to infer (loop) invariants ?

Fundamental Challenge of Program Analysis

- Key issue for any analysis or verification
- Many algorithms/heuristics
- See Suzuki & Ishihata, POPL 1977
- Most formalizable in framework of **Abstract Interpretation**

Abstract Interpretation

"A systematic basis for *approximating* the semantics of programs"

- Deep and broad area
- Rich theory
- Profound practical impact

We look at a tiny slice

- In context of algorithmic verification of **IMP**

IMP: A Small Imperative Language

Recall the syntax of IMP

```
data Com = Var `:=` Expr              -- assignment
         | Com `;`  Com               -- sequencing
         | Assume   Exp               -- assume
         | Com `|`  com               -- branch
         | While Pred Exp Com         -- loop
```

Note

We have thrown out `If` and `Skip` using the abbreviations:

```
Skip         == Assume True
If e c1 c2   == (Assume e; c1) | (Assume (!e); c2)
```

IMP: Operational Semantics

States

A `State` is a map from `Var` to the set of `Values`

```
type State = Map Var Value
```

IMP: Operational Semantics

Transition Relation

A subset of `State` × `Com` × `State` formalized by

- `eval s c == [s'` | command c transitions state s to s'`]`

```
eval                      :: State -> Com -> [State]
eval s (Assume e)   = if eval s e then [s] else []
eval s (x  :=  e)   = [ add x (eval s e) s ]
eval s (c1 ; c2)     = [s2 | s1 <- eval s c1, s2 <- eval s'
eval s (c1 | c2)     = eval s c1 ++ eval s c2
eval s w@(Whle e c) = eval s $ Assume !e | (Assume e; c; w)
```

IMP: Axiomatic Semantics

State Assertions

- An assertion `P` is a `Predicate` over the set of program variables.
- An assertion corresponds to a **set of states**

```
states P = [s | eval s P == True]
```

IMP: Axiomatic Semantics

Describe execution via **Predicate Transformers**

Strongest Postcondition

```
SP :: Pred -> Com -> Pred
```

`SP P c` : States **reachable from** `P` by executing `c`

```
states (SP P c) == [s' | s <- states P, s' <- eval s c]
```

IMP: Axiomatic Semantics

Describe execution via **Predicate Transformers**

Weakest Precondition

```
WP :: Com -> Pred -> Pred
```

`WP c Q` : States that **can reach** `Q` by executing c

```
states (WP c Q)` = [s | s' <- eval s c, eval s' Q ]
```

Strongest Postcondition

`SP P c` : States **reachable from** `P` by executing `c`

```
SP                  :: Pred -> Com -> Pred

SP P (Assume e) = P `&&` e

SP P (x := e)    = Exists x'. P[x'/x] `&&`  x `==` e[x'/x]

SP P (c1 ; c2)   = SP (SP P c1) c2

SP P (c1 | c2)  = SP P c1 `||` SP p c2

SP P w@(W e c)   = SP s (Assume !e | (Assume e; c; w))
```

- **Uh Oh!** last case is non-terminating . . .

Weakest Precondition

`WP c Q` : States that **can reach** `Q` by executing `c`

```
WP                :: Com -> Pred -> Pred

WP (Assume e) Q = e `=>` Q

WP (x := e)   Q = Q[e/x]

WP (c1 ; c2)  Q = WP c1 (WP c2 Q)

WP (c1 | c2)  Q = WP c1 Q `&&` WP c2 Q

WP w@(W e c)  Q = WP (Assume !e | (Assume e; c; w)) Q
```

- **Uh Oh!** last case is non-terminating ...

IMP: Verification (Suspend disbelief regarding loops)

Goal: Verify Hoare-Triples

Given

- `c` command
- `P` precondition
- `Q` postcondition

Prove

- **Hoare-Triple** $\{$`P`$\}$ `c` $\{$`Q`$\}$ which denotes

```
forall s s'. if s  `in` (states P) &&
                s' `in` (eval s c)
             then
                s' `in` (states Q)
```

Verification Strategy

(For a moment, suspend disbelief regarding loops)

1. Compute Verification Condition (VC)
   - `(SP P c) => Q`
   - `P => (WP c Q)`
2. Use SMT Solver to Check VC is Valid

Verification Strategy

1. Compute Verification Condition (VC)
   - `(SP P c) => Q`
   - `P => (WP c Q)`
2. Use SMT Solver to Check VC is Valid

Problem: Pesky Loops

- Cannot compute `WP` or `SP` for `While b c` ...
- ... Require **invariants**

Next: Lets **infer** invariants by **approximation**

Approximate Verification Strategy

0. Compute **Over-approximate** Postcondition `SP#` s.t.
   - `(SP P c) => (SP# P c)`
1. Compute Verification Condition (VC)
   - `(SP# P c) => Q`
2. Use SMT Solver to Check VC is Valid
   - If so, `{P} c {Q}` holds by **Consequence Rule**

Key Requirement

- Compute `SP#` **without** computing `SP` ...
- But **guaranteeing** over-approximation

What Makes Loops Special?

Why different from other constructs? Let

- `c` be a loop-free (i.e. has no `While` inside it)
- `W` be the loop `While b c`

Loops as Limits

Inductively define the *infinite* sequence of loop-free `Com`

```
W_0   = Skip

W_1   = W_0 | Assume b; c; W_0

W_2   = W_1 | Assume b; c; W_1
.
.
.
W_i+1 = W_i | Assume b; c; W_i
.
.
.
```

Loops as Limits

Intuitively

- `W_i` is the loop **unrolled upto** `i` times
- `W == W_0 | W_1 | W_2 | ...`

Formally, we can prove (**exercise**)

```
1. eval s W  == eval s W_0 ++ eval s W_1 ++ ...

2. SP P W    == SP P W_0   || SP P W_1   || ...

3. WP W Q    == WP W_0 Q   && WP W_1 Q   && ...
```

So what? Still cannot **compute** `SP` or `WP` ...!

Loops as Limits

So what? Still cannot **compute** SP or WP ... but notice

```
SP P W_i+1 == SP P (W_i | assume b; c; W_i)
           == SP P W_i  || SP (SP P (assume b; c)) W_i
           <= SP P W_i
```

That is, SP P W_i form an **increasing chain**

```
SP P W_0 => SP P W_1 => ... => SP P W_i => ...
```

... **Problem:** Chain does not converge! *ONION RINGS*

Approximate Loops as Approximate Limits

To find `SP#` such that `SP P c => SP# P c`, we compute chain

```
SP# P W_0 => SP# P W_1 => ... => SP# P W_i => ...
```

where each `SP#` is over-approximates the corresponding `SP`

```
for all i. SP P W_i => SP# P W_i
```

and the chain of `SP#` chain converges to a **fixpoint**

```
exists j. SP# P W_j+1 == SP# P W_j
```

This magic `SP# P W_j+1` is the loop invariant, and

```
SP# P W == SP# P W_j
```

Approximating Loops

Many Questions Remain Around Our Strategy

How to compute `SP#` so that we can ensure

1. **Convergence** to a fixpoint ?
2. Result is an **over-approximation** of `SP` ?

Answer: Abstract Interpretation

"Systematic basis for **approximating the semantics** of programs"

Abstract Interpretation

Plan

1. Simple language of arithmetic expressions
2. IMP
3. Predicate Abstraction (AI using SMT)

A Language of Arithmetic

Our language, just has numbers and multiplication

A Language of Arithmetic: Syntax

```
data AExp = N Int | AExp `Mul` AExp
```

Example Expressions

```
N 7
```

```
N 7 `Mul` N (-3)
```

```
N 0 `Mul` N 7 `Mul` N (-3)
```

Concrete Semantics

To define the **(concrete)** or **exact** semantics, we need

```
type Value = Int
```

and an `eval` function that maps `AExp` to `Value`

```
eval                :: AExp -> Value
eval (N n)          = n
eval (Mul e1 e2) = mul (eval e1) (eval e2)

mul n m             = n * m
```

Signs Abstraction

Suppose that we only care about the **sign** of the number.

Can define an *abstract* semantics

1. Abstract Values
2. Abstract Operators
3. Abstract Evaluators

Signs Abstraction: Abstract Values

Abstract values just preserve the **sign** of the number

```
data Value# = Neg | Zero | Pos
```

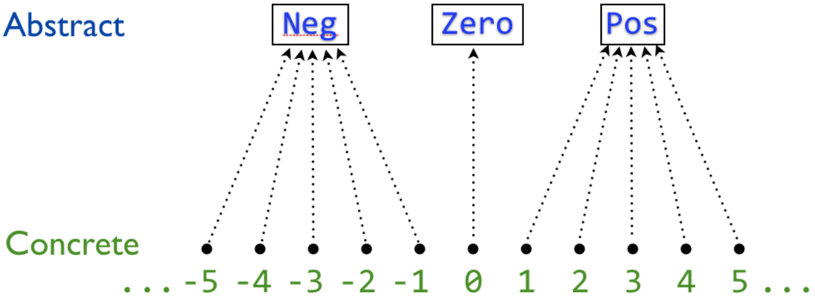

Figure: Abstract and Concrete Values

Signs Abstraction: Abstract Evaluator

Abstract evaluator just uses sign information

```
eval#                :: AExp -> Value#
eval# | n > 0        = Pos
      | n < 0        = Neg
      | otherwise    = Zero
eval# (Mul e1 e2) = mul# (eval# e1) (eval# e2)
```

Signs Abstraction: Abstract Evaluator

`mul#` is the **abstract multiplication** operators

```
mul#                :: Value# -> Value# -> Value#
mul# Zero _          = Zero
mul# _    Zero       = Zero
mul# Pos Pos         = Pos
mul# Neg Neg         = Pos
mul# Pos Neg         = Neg
mul# Neg Pos         = Neg
```

Connecting the Concrete and Abstract Semantics

Theorem For all `e :: AExp` we have

1. `(eval e)` $>$ `0` iff `(eval# e) = Pos`
2. `(eval e)` $<$ `0` iff `(eval# e) = Neg`
3. `(eval e) = 0` iff `(eval# e) = Zero`

Proof By induction on the structure of `e`

- Base Case: `e == N n`
- Ind. Step: Assume above for `e1` and `e2` prove for `Mul e1 e2`

Relating the Concrete and Abstract Semantics

Next, let us **generalize** what we did into a **framework**

- Allows us to use **different** `Value#`
- Allows us to get **connection theorem** by construction

Key Idea: Provide Abstraction Function α

We only have to provide connection between `Value` and `Value#`

```
alpha :: Value -> Value#
```

Key Idea: Provide Abstraction Function α

We only have to provide connection between `Value` and `Value#`

```
alpha :: Value -> Value#
```

For *signs* abstraction

```
alpha n | n > 0       = Pos
        | n < 0       = Neg
        | otherwise   = Zero
```

Key Idea: α induces Concretization γ

Given `alpha :: Value -> Value#`

we get for free a **concretization function**

```
gamma     :: Value# -> [Value]
gamma v# =  [ v | (alpha v) == v# ]
```

For *signs* abstraction

```
gamma Pos  == [1,2..]
gamma Neg  == [-1,-2..]
gamma Zero == [0]
```

Key Idea: α induces Abstract Operator

Given `alpha :: Value -> Value#`

we get for free a **abstract operator**

```
op# x# y# = alpha (op (gamma x#) (gamma y#))
```

(actually, there is some *cheating* above...can you spot it?)

Key Idea: α induces Abstract Operator

Given `alpha :: Value -> Value#`

we get for free a **abstract operator**

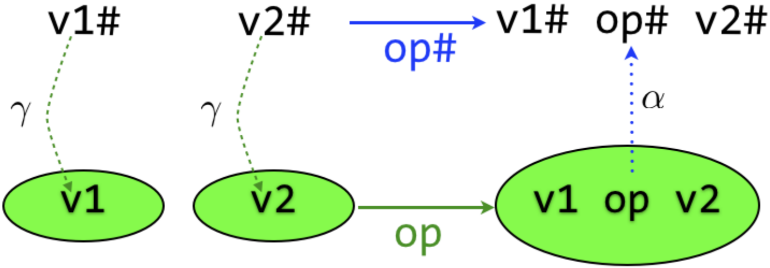

Figure: Abstract Operator

Key Idea: α induces Abstract Evaluator

Given `alpha :: Value -> Value#`

we get for free a **abstract evaluator**

```
eval#              :: AExp -> Value#
eval# (N n)        = (alpha n)
eval# (Op e1 e2) = op# (eval# e1) (eval# e2)
```

Key Idea: α induces Connection Theorem

Given `alpha :: Value -> Value#`

we get for free a **connection theorem**

Theorem For all `e::AExp` we have

1. `(eval e) in gamma (eval# e)`
2. `alpha(eval e) = (eval# e)`

Proof Exercise (same as before, but generalized)

Key Idea: α induces Connection Theorem

Given `alpha :: Value -> Value#`

we get for free a **connection theorem**

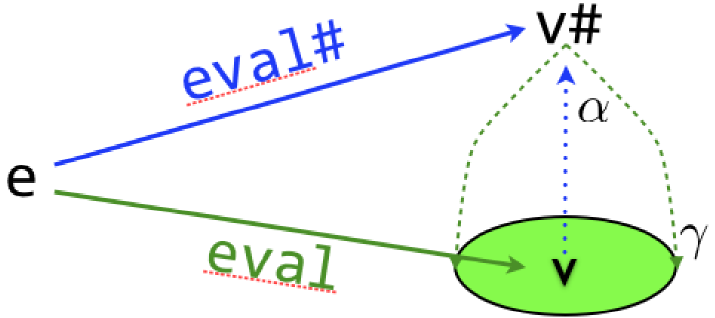

Figure: Connection Theorem

Our First Abstract Interpretation

Given: Language AExp and Concrete Semantics

```
data AExp
data Value

op    :: Value -> Value -> Value
eval  :: AExp  -> Value
```

Given: Abstraction

```
data Value#
alpha :: Value -> Value#
```

Our First Abstract Interpretation

Obtain for free: Abstract Semantics

```
op#   :: Value# -> Value# -> Value#
eval# :: AExp -> Value#
```

Obtain for free: Connection

Theorem: Abstract Semantics **approximates** Concrete Semantics

Our Second Abstract Interpretation

Let us extend `AExp` with new operators

- **Negation**
- Addition
- Division

AExp with Unary Negation

Extended Syntax

```
data AExp = ... | Neg AExp
```

Extended Concrete Semantics

```
eval (Neg e) = neg (eval e)
```

AExp with Unary Negation

Derive Abstract Operator

```
neg# :: Value# -> Value#
neg# = alpha . neg . gamma
```

Which is equivalent to (if you do the math)

```
neg# Pos  = Neg
neg# Zero = Zero
neg# Neg  = Pos
```

Theorem holds as before!

Our Third Abstract Interpretation

Let us extend `AExp` with new operators

- Negation
- **Addition**
- Division

AExp with Addition

Extended Syntax

```
data AExp = ... | Add AExp AExp
```

Extended Concrete Semantics

```
eval (Add e1 e2) = plus (eval e1) (eval e2)
```

AExp with Addition

Derive Abstract Operator

```
plus#         :: Value# -> Value# -> Value#
plus# v1# v2# = alpha (plus (gamma v1#) (gamma v2#))
```

That is,

```
plus# Zero v# = v#
plus# Pos Pos = Pos
plus# Neg Neg = Neg
```

but . . .

```
plus# Pos Neg = ???
plus# Neg Pos = ???
```

Problem: Require Better Abstract Values

Need new value to represent **union of** *positive* and *negative*

- `T` (read: **Top**), denotes **any** integer

Now, we can define

```
plus# Zero v# = v#
plus# Top  v# = Top
plus# Pos Pos = Pos
plus# Neg Neg = Neg
plus# Pos Neg = Top
plus# Neg Pos = Top
```

Semantics is now Over-Approximate

Notice that now,

```
eval  (N 1 `Add` N 2 `Add` (Neg 3)) == 0
eval# (N 1 `Add` N 2 `Add` (Neg 3)) == T
```

That is, we have **lost all information** about the sign!

- This is **good**
- Exact semantics **not computable** for real PL!

Our Fourth Abstract Interpretation

Let us extend `AExp` with new operators

- Negation
- Addition
- **Division**

AExp with Division

Extended Syntax

```
data AExp = ... | Div AExp AExp
```

Extended Concrete Semantics

```
eval (Add e1 e2) = div (eval e1) (eval e2)
```

AExp with Division: Abstract Semantics

How to define

```
div# v# Zero = ?
```

Need new value to represent **empty set** of integers

- _|_ (read: **Bottom**), denotes **no** integer
- Abstract operator on _|_ returns _|_
- Wait, this is getting rather *ad-hoc* ...
- Need more **structure** on `Value#`

Abstract Values Form Complete Partial Order

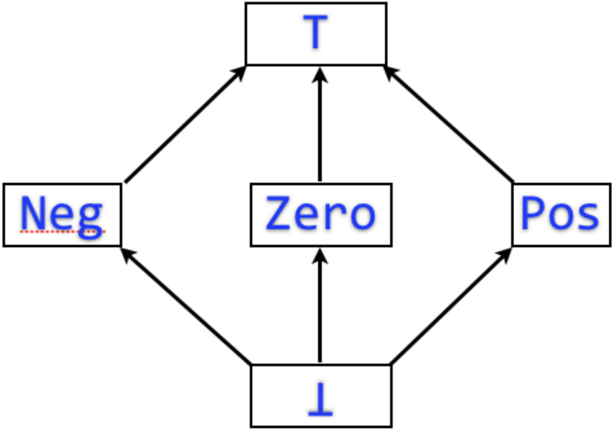

Figure: `Value#` Forms Complete Partial Order

Abstract Values Form Complete Partial Order

```
-- Partial Order
(<=) :: Value# -> Value# -> Bool

-- Greatest Lower Bound
glb :: Value# -> Value# -> Value#

-- Least Upper Bound
lub :: Value# -> Value# -> Value#
```

`leq v1# v2#` means `v1#` corresponds to **fewer** concrete values than `v2#`

Examples

- `leq` $\perp$ `Zero`
- `leq Pos Top`

Abstract Values: Least Upper Bound

```
forall v1# v2#.  v1# <= lub v1# v2#
forall v1# v2#.  v2# <= lub v1# v2#
forall v      .  if v1# <= v && v2# <= v then lub v1# v2# <
```

Examples

- (lub _|_ Zero) == Zero
- (lub Neg Pos) == Top

~~~~~
~~~~~

Abstract Values: Greatest Lower Bound

```
forall v1# v2#.  glb v1# v2# <= v1#
forall v1# v2#.  glb v1# v2# <= v2#
forall v      .  if v <= v1# && v <= v2# then v <= glb v1#
```

Examples

- `(glb Pos Zero) == _|_`
- `(lub Top Pos) == Pos`

Key Idea: α and CPO induces Concretization γ

Given

- α :: `Value -> Value#`
- $\sqsubseteq$ :: `Value# -> Value# -> Bool`

We get for free a **concretization function**

- γ :: `Value# -> [Value]`

```
gamma     :: Value# -> [Value]
gamma v# =  [ v | (alpha v) <= v# ]
```

Theorem `v1#` $\sqsubseteq$ `v2#` iff `(gamma v1#)` $\subseteq$ `(gamma v2#)`

That is,

- `v1#` $\sqsubseteq$ `v2#` means `v1#` represents **fewer** `Value` than `v2#`

Key Idea: α and CPO induces α over `[Value]`

We can now lift α to work on **sets** of values

```
alpha    :: [Value] -> Value#
alpha vs = lub [alpha v | v <- vs]
```

For example

```
alpha [3, 4]  == Pos
alpha [-3, 4] == Top
alpha [0]     == Zero
```

Key Idea: α + CPO induces Abstract Operator

Given

- α :: `Value -> Value#`
- $\sqsubseteq$ :: `Value# -> Value# -> Bool`

We get for free a **abstract operator**

```
op# x# y# = alpha [op x y | x <- gamma x#, y <- gamma y#]
```

i.e., `lub` of results of point-wise concrete operator (*no cheating*!)

For example

```
plus# Pos Neg
  == alpha [x + y | x <- gamma Pos, y <- gamma Neg]
  == alpha [x + y | x <- [1,2..]   , y <- [-1,-2..]]
  == alpha [0,1,-1,2,-2..]
  == Top
```

Key Idea: α + CPO induces Abstract Operator

Given `alpha :: Value -> Value#`

we get for free a **abstract operator**

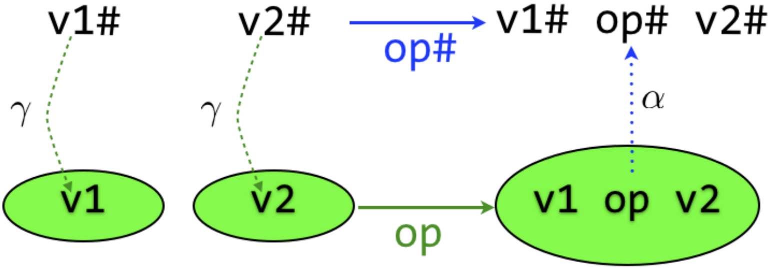

Figure: Abstract Operator

Key Idea: α + CPO induces Evaluator

As before, we get for free a **abstract evaluator**

```
eval#                :: AExp -> Value#
eval# (N n)          = (alpha n)
eval# (Op e1 e2) = op# (eval# e1) (eval# e2)
```

Key Idea: α + CPO induces Evaluator

And, more importantly, the semantics connection

Theorem For all `e::AExp` we have

1. `(eval e)` $\in$ `gamma (eval# e)`
2. `alpha (eval e)` $\sqsubseteq$ `(eval# e)`

Over-Approximation

In bare `AExp` we had **exact** abstract semantics

- `alpha (eval e)` $=$ `(eval# e)`

Now, we have **over-approximate** abstract semantics

- `alpha (eval e)` $\sqsubseteq$ `(eval# e)`

That is, information is lost.

Next Time: Abstract Interpretation For IMP

So far, abstracted values for `AExp`

- Concrete `Value` = `Int`
- Abstract `Value#` = `Signs`

Next time: apply these ideas to IMP

- Concrete `Value` = `State` at program points
- Abstract `Value#` = ???

Abstract Semantics yields **loop invariants**