Abstract interpretation
(An introduction)

Ralf Lämmel
Facebook, London
http://www.softlang.org/rlaemmel:home
Dependencies:
- Basics of interpretation
- Denotational semantics
A program with an optimization opportunity

```java
{ 
  ... 
  y = x * x + 42;
  if (y < 0) 
    y = -y;
  ... 
}
```

The variable y must be positive at this point. Thus, the condition must be false. No need to test the condition. No need to keep the code in the then branch.
Abstract operations (on signs)

<table>
<thead>
<tr>
<th></th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Pos</td>
<td>Zero</td>
<td>Pos</td>
<td>?</td>
</tr>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
<td>Zero</td>
</tr>
<tr>
<td>Pos</td>
<td>Neg</td>
<td>Zero</td>
<td>Pos</td>
<td>?</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>Zero</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Neg</td>
<td>Neg</td>
<td>Neg</td>
<td>?</td>
</tr>
<tr>
<td>Zero</td>
<td>Neg</td>
<td>Zero</td>
<td>Pos</td>
<td>?</td>
</tr>
<tr>
<td>Pos</td>
<td>?</td>
<td>Pos</td>
<td>Pos</td>
<td>?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Neg</th>
<th>Zero</th>
<th>Pos</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>?</td>
<td>True</td>
<td>True</td>
<td>?</td>
</tr>
<tr>
<td>Zero</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>?</td>
</tr>
<tr>
<td>Pos</td>
<td>False</td>
<td>False</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

**Key idea**: go from concrete to abstract interpretation such that properties (abstract values such as signs) are computed -- as opposed concrete values.
Factoring out concrete interpretation

execute :: Stmt → StoreT
execute Skip = skip'
execute (Assign x e) = assign' x (evaluate e)
execute (Seq s1 s2) = seq' (execute s1) (execute s2)
execute (If e s1 s2) = if' (evaluate e) (execute s1) (execute s2)
execute (While e s) = while' (evaluate e) (execute s)
...

skip' :: StoreT
skip' = id
assign' :: String → StoreO → StoreT
assign' x f m = insert x (f m) m
seq' :: StoreT → StoreT → StoreT
seq' = flip (.)
if' :: StoreO → StoreT → StoreT → StoreT
if' f g h m = let Right v = f m in if v then g m else h m
while' :: StoreO → StoreT → StoreT
while' f g = fix h where h t = if' f (seq' g t) skip'
...

This "compositional schema" should be parametrized in an algebra.

Such functions should be grouped as an "algebra".
The type of semantic algebras

```haskell
-- Aliases to shorten function signatures
type Trafo sto = sto → sto -- Store transformation
type Obs sto val = sto → val -- Store observation
-- The signature of algebras for interpretation
data Alg sto val = Alg {
    skip' :: Trafo sto,
    assign' :: String → Obs sto val → Trafo sto,
    seq' :: Trafo sto → Trafo sto → Trafo sto,
    if' :: Obs sto val → Trafo sto → Trafo sto → Trafo sto,
    while' :: Obs sto val → Trafo sto → Trafo sto → Trafo sto,
    intconst' :: Int → Obs sto val,
    var' :: String → Obs sto val,
    unary' :: UOp → Obs sto val → Obs sto val,
    binary' :: BOp → Obs sto val → Obs sto val → Obs sto val
}
```

© 2017-2020 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
The compositional scheme

interpret :: Alg sto val → Stmt → sto
interpret a = execute

where

    \[\text{—— Compositional interpretation of statements}\]
    execute Skip = skip' a
    execute (Assign x e) = assign' a x (evaluate e)
    execute (Seq s1 s2) = seq' a (execute s1) (execute s2)
    execute (If e s1 s2) = if' a (evaluate e) (execute s1) (execute s2)
    execute (While e s) = while' a (evaluate e) (execute s)

    \[\text{—— Compositional interpretation of expressions}\]
    evaluate (IntConst i) = intconst' a i
    evaluate (Var n) = var' a n
    evaluate (Unary o e) = unary' a o (evaluate e)
    evaluate (Binary o e1 e2) = binary' a o (evaluate e1) (evaluate e2)
**An algebra for the standard interpreter**

```haskell
-- 12.5 Abstract Interpretation 49

type Value = Either Int Bool

type Store = Map String Value

algebra :: Alg Store Value

algebra = a where a = Alg {
    skip' = id,
    assign' = \n n f m \rightarrow \text{insert} \ n \ (f \ m) \ m,
    seq' = flip (.),
    if' = \lambda f g h \ m \rightarrow \text{let} \ (\text{Right} \ b) = f \ m \ \text{in} \ \text{if} \ b \ \text{then} \ g \ m \ \text{else} \ h \ m,
    while' = \lambda f g \rightarrow \text{fix} \ (\lambda x \rightarrow \text{if'} \ a \ f \ (\text{seq'} \ a \ g \ x) \ \text{(skip'} \ a))
    intconst' = \lambda i \rightarrow \text{const} \ (\text{Left} \ i),
    var' = \lambda n m \rightarrow m!n,
    unary' = \lambda o f m \rightarrow 
        \text{case} \ (o, f m) \ \text{of}
            (\text{Negate}, \text{Left} \ i) \rightarrow \text{Left} \ (\text{negate} \ i)
            (\text{Not}, \text{Right} \ b) \rightarrow \text{Right} \ (\text{not} \ b),
    binary' = \lambda o f g m \rightarrow ...
}
```

Thus, the algebra commits to the sum of `Int` and `Bool` for values (line 1), and to maps from strings to values for stores (line 2), and it designates the usual operations for combining meanings. For instance, if-statements are eventually handled by a dispatch on a condition’s two possible values, `True` and `False` (line 8).

12.5.4 Abstract Domains

An abstract interpretation devises abstract domains to analyze programs statically, as opposed to a description of the precise semantics in terms of its so-called concrete domains. For instance, an abstract interpretation for type checking would use abstract domains as follows:

```haskell
data Type = IntType | BoolType
```

Instead of values type

```haskell
VarTypes = Map String Type
```

Instead of stores

That is, abstract interpretation should compute variable-to-type maps as opposed to proper stores, i.e., variable-to-value maps. The idea is then that the semantic combinators on abstract domains are defined similarly to those for the concrete domains. In algebraic terms, we use (chain-) complete partial orders (CCPO or CPO). An abstract interpretation for sign detection would use abstract domains as follows:

```haskell
data Sign = Zero | Pos | Neg | BottomSign | TopSign

data CpoBool = ProperBool Bool | BottomBool | TopBool

type Property = Either Sign CpoBool

type VarProperties = Map String Property
```
Abstract domains for sign detection

Abstract Interpretation

Haskell module Language.BIPL.Algebra.StandardInterpreter

```haskell
12.5 Abstract Interpretation 49

Illustration 12.27
(An algebra for interpretation)

1. type Value = Either Int Bool

2. type Store = Map String Value

3. algebra :: Alg Store Value

4. algebra = a
   where

5.   skip' = id, 

6.   assign' =
      \( n : f m \mapsto m \mapsto \) 

7.   seq' = flip (.), 

8.   if' =
      let 
          (Right b) = f m
      in 
          if
          b
          then 
          gm
          else 
          h m

9.   while' =
      flip (.), 

10. intconst' =

11. var' =

12. unary' =

13. case

Thus, the algebra commits to the sum of `Int` and `Bool` for values (line 1), and to maps from strings to values for stores (line 2), and it designates the usual operations for combining meanings. For instance, if-statements are eventually handled by a dispatch on a condition's two possible values, `True` and `False` (line 8).

12.5.4 Abstract Domains

An abstract interpretation devises abstract domains to analyze programs statically, as opposed to a description of the precise semantics in terms of its so-called concrete domains. For instance, an abstract interpretation for type checking would use abstract domains as follows:

```haskell
data Type = IntType | BoolType

Instead of values

data VarTypes = Map String Type

Instead of stores

That is, abstract interpretation should compute variable-to-type maps as opposed to proper stores, i.e., variable-to-value maps. The idea is then that the semantic combinators on abstract domains are defined similarly to those for the concrete domains.

In algebraic terms, we use (chain-) complete partial orders (CCPO or CPO). An abstract interpretation for sign detection would use abstract domains as follows:

```haskell
data Sign = Zero | Pos | Neg | BottomSign | TopSign
data CpoBool = ProperBool Bool | BottomBool | TopBool
type Property = Either Sign CpoBool
type VarProperties = Map String Property
```
Signs as numbers

```
data Sign = Zero | Pos | Neg | BottomSign | TopSign

instance Num Sign
  where
    fromInteger n
    | n > 0 = Pos
    | n < 0 = Neg
    | otherwise = Zero

  TopSign + _ = TopSign
  _ + TopSign = TopSign
  BottomSign + _ = BottomSign
  _ + BottomSign = BottomSign
  Zero + Zero = Zero
  Zero + Pos = Pos
  ...
```
Signs as CPOs

instance CPO Sign where
  pord x y | x == y = True
  pord BottomSign _ = True
  pord _ TopSign = True
  pord _ _ = False
lub x y | x == y = x
lub BottomSign x = x
lub x BottomSign = x
lub _ _ = TopSign

instance Bottom Sign where
  bottom = BottomSign
Implement the more precise option shown in Fig. 12.5.

```
<table>
<thead>
<tr>
<th>With less precision</th>
<th>With more precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neg</td>
<td>Non-Neg</td>
</tr>
<tr>
<td>Zero</td>
<td>Non-Pos</td>
</tr>
<tr>
<td>Pos</td>
<td>Pos</td>
</tr>
</tbody>
</table>
```

For an abstract interpretation to be sound with regard to a given standard semantics or program analysis more generally, in terms of precision of results versus time and space complexity required.

Because we want to indicate that one can make a trade-off in abstract interpretation of analysis. We show two options for the abstract domain of signs in the figure (see below for details), whereas the greatest element \( \top \) is the initial element for any sort of approximative, fixed point-based analysis (lines 3–16), paralleling the standard instance for type `Sign` from `Int` (lines 5–8) is the explicit manifestation of abstraction.

```haskell
class Sign a where
  fromInt :: Int -> a
```

The excerpts given here illustrate the following aspects of signs:

- Signs form a partial order, and there are least and greatest elements; see the in-
- Signs are "abstract" numbers; Haskell's library type class
- Signs as abstract numbers
- Precision of abstract domains

© 2017-2020 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
An algebra for sign detection

```haskell
type Property = Either Sign CpoBool

type VarProperties = Map String Property

algebra :: Alg VarProperties Property
algebra = a where a = Alg {
    'skip' = id,
    'assign' = \n f m \rightarrow insert n (f m) m,
    'seq' = flip (.),
    'if' = \ f g h m \rightarrow
        let Right b = f m in
        case b of
            (ProperBool True) \rightarrow g m
            (ProperBool False) \rightarrow h m
            BottomBool \rightarrow bottom
            TopBool \rightarrow g m `lub` h m,
    'while' = \ f g \rightarrow fix' (\ x \rightarrow if' a f (x . g) id) (const bottom),
    'intconst' = \ i \rightarrow const (Left (fromInteger (toInteger i))),
    'var' = \ n m \rightarrow m!n,
    'unary' = \ o f m \rightarrow
        case (o, f m) of
            (Negate, Left s) \rightarrow Left (negate s)
            (Not, Right b) \rightarrow Right (cpoNot b),
    'binary' = \ o f g m \rightarrow ...
}
```

Just like in the standard interpreter

Pick then vs. else, if possible

WTF!?

"Trivial"
Fixed point computations

Fixed points in denotational interpreter

\[
\text{fix} :: (a \to a) \to a \\
\text{fix } f = f (\text{fix } f)
\]

Fixed points in abstract interpreter

\[
\text{fix'} :: \text{Eq } a \Rightarrow ((a \to a) \to a \to a) \to (a \to a) \to a \to a \\
\text{fix'} h i x = \text{limit} \ (\text{iterate } h \ i) \\
\quad \text{where} \ \text{limit} \ (b_1:b_2:bs) = \text{if} \ b_1 \ x == b_2 \ x \ \text{then} \ b_1 \ x \ \text{else} \ \text{limit} \ (b_2:bs)
\]

The algebra member

\[
\text{while'} = \lambda f g \rightarrow \text{fix'} (\lambda x \rightarrow \text{if' } a \ f \ (x \ . \ g) \ \text{id}) \ (\text{const bottom}),
\]

N.B.: The iterate function builds an infinite list:

\[
\text{iterate} :: (a \to a) \to a \to [a]
\]
Two variations on factorial

V1

```plaintext
y = 1;
i = 1;
while (i <= x) {
  y = y * i;
i = i + 1;
}
```

V2

```plaintext
y = 1;
while (x >= 2) {
  y = y * x;
x = x - 1;
}
```

In both cases, sign analysis should determine that 
**y is positive at the end of the program** (past the while loop).
Two attempts at sign analysis

A basic, less useful analysis

- interpret BasicAnalysis.algebra facv1 (fromList [('x', Left Pos)])
  fromList [('i', Left Pos), ('x', Left Pos), ('y', Left Pos)]

- interpret BasicAnalysis.analysis facv2 (fromList [('x', Left Pos)])
  fromList [('x', Left TopSign), ('y', Left TopSign)]

A refined, more useful analysis

- interpret RefinedAnalysis.algebra facv1 (fromList [('x', Left Pos)])
  fromList [('i', Left Pos), ('x', Left Pos), ('y', Left Pos)]

- interpret RefinedAnalysis.algebra facv2 (insert "x" (Left Pos) empty)
  fromList [('x', Left TopSign), ('y', Left Pos)]

The refined analysis leverages a more nuanced approach towards combining then and else branches in ifs (not discussed here in detail).
Online resources

YAS’ GitHub repository contains all code.
YAS (Yet Another SLR (Software Language Repository))
http://www.softlang.org/yas
See Haskell-based concrete and abstract interpreters of BIPL.

The Software Languages Book
http://www.softlang.org/book
The book covers various metaprogramming techniques.