Programming Language Theory

Program Analysis

Ralf Lämmel
Program analysis--what for?

- Compilation
  - Optimization
- IDE
  - Find programming errors
  - Check pre-conditions of refactoring
- Re-engineering
  - Dead-code elimination

We are particularly interested in program analysis of the kind that gives a reliable statement about the execution of a program.
Example 1

**Constant propagation**: determine whether an expression always evaluates to a constant and if so determine that value.

Program: \( x := 5; y := x \times x + 25 \)

Analysis: \( y \) evaluates to 50.

Optimized program: \( x := 5; y := 50 \)
Example 2

**Sign analysis**: determine the sign of an expression.

Program: \[ y := x \times x + 25; \text{while } y \leq 0 \text{ do } \cdots \]

Analysis: \[ y \text{ is always positive.} \]

Optimized program: \[ y := x \times x + 25 \]
Classes of program analysis

- **Forward analyses**: given a property of the input, we determine the properties of the result.

- **Backward analyses**: given a property of the result, we determine the properties the input should have.

Detection of signs or constant propagation

Derivation of weakest pre-conditions
Program analysis and the halting problem

program analysis
≡
how to get information about programs without running them

unsolvability of the halting problem

↓

tell the truth
but not the complete truth
Detection of Signs Analysis
(Motivation)

**Example**

What is the sign of \((0 - 5) \times 3\)?

\[
\begin{array}{c}
(0 \quad -5) \\
\underline{\text{ZERO \quad POS}} \\
\underline{\text{NEG \quad POS}} \\
\underline{\text{NEG}}
\end{array}
\]

**Required rules for calculating with signs**

\[
\begin{array}{|c|ccc|}
\hline
\times & \text{POS} & \text{ZERO} & \text{NEG} \\
\hline
\text{POS} & \text{POS} & \text{ZERO} & \text{NEG} \\
\text{ZERO} & \text{ZERO} & \text{ZERO} & \text{ZERO} \\
\text{NEG} & \text{NEG} & \text{ZERO} & \text{POS} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|ccc|}
\hline
- & \text{POS} & \text{ZERO} & \text{NEG} \\
\hline
\text{POS} & \text{ANY} & \text{POS} & \text{POS} \\
\text{ZERO} & \text{NEG} & \text{ZERO} & \text{POS} \\
\text{NEG} & \text{POS} & \text{NEG} & \text{ANY} \\
\text{ANY} & \text{ANY} & \text{ANY} & \text{ANY} \\
\hline
\end{array}
\]
The **sign** as a “‘property’” of numbers

Again, we use Hasse diagrams for the partial orders (in fact, complete lattices) at hand.
The **sign** as a “property” of numbers

Our properties can aspire to different degrees of precision.
From denotational semantics to program analysis

- Replace numbers: \( Z \) by properties: \( P_Z \)
- Replace truth values: \( T \) by properties: \( P_T \)
- Replace states: \( \text{State} = \text{Var} \rightarrow Z \) by property states: \( \text{PState} = \text{Var} \rightarrow P_Z \)

Replace semantic functions on values and states by semantic functions on properties and property states.
From denotational semantics to program analysis

Direct style denotational semantics:

- $\mathcal{A}: \text{Aexp} \rightarrow \text{State} \rightarrow \mathbb{Z}$
- $\mathcal{B}: \text{Bexp} \rightarrow \text{State} \rightarrow \mathbb{T}$
- $S_{ds}: \text{Stm} \rightarrow (\text{State} \hookrightarrow \text{State})$
From denotational semantics to program analysis

Direct style denotational semantics:

• \( A : Aexp \to State \to Z \)
• \( B : Bexp \to State \to T \)
• \( S_{ds} : Stm \to (State \hookrightarrow State) \)

Forward program analysis:

• \( F_A : Aexp \to PState \to P_Z \)
• \( F_B : Bexp \to PState \to P_T \)
• \( F_S : Stm \to PState \to PState \)
From denotational semantics to program analysis

Forward program analysis:

- $\mathcal{F}A : \text{Aexp} \rightarrow \text{PState} \rightarrow \text{P}_Z$
- $\mathcal{F}B : \text{Bexp} \rightarrow \text{PState} \rightarrow \text{P}_T$
- $\mathcal{F}S : \text{Stm} \rightarrow \text{PState} \rightarrow \text{PState}$

Backward program analysis:

- $\mathcal{B}A : \text{Aexp} \rightarrow \text{P}_Z \rightarrow \text{PState}$
- $\mathcal{B}B : \text{Bexp} \rightarrow \text{P}_T \rightarrow \text{PState}$
- $\mathcal{B}S : \text{Stm} \rightarrow \text{PState} \rightarrow \text{PState}$
Application of a forward analysis

- Define a suitable initial property state.
- Compute resulting property state with the program analysis.

Express assumptions about program variables in the beginning.

Requires special fixed-point approach to guarantee termination!
Let’s define a sign analysis.

Direct style denotational semantics:

\[
\begin{align*}
\text{State} &= \text{Var} \rightarrow \mathbb{Z} \\
\mathcal{A} : \text{Aexp} &\rightarrow \text{State} \rightarrow \mathbb{Z} \\
\mathcal{B} : \text{Bexp} &\rightarrow \text{State} \rightarrow \mathbb{I} \\
\mathcal{S}_{ds} : \text{Stm} &\rightarrow (\text{State} \leftrightarrow \text{State})
\end{align*}
\]

Detection of signs analysis:

\[
\begin{align*}
\text{PState} &= \text{Var} \rightarrow \text{Sign} \\
\mathcal{S}_A : \text{Aexp} &\rightarrow \text{PState} \rightarrow \text{Sign} \\
\mathcal{S}_B : \text{Bexp} &\rightarrow \text{PState} \rightarrow \mathbb{I} \\
\mathcal{S}_S : \text{Stm} &\rightarrow \text{PState} \rightarrow \text{PState}
\end{align*}
\]
Analysis of arithmetic expressions

\[ SA : \text{Aexp} \rightarrow \text{PState} \rightarrow \text{Sign} \]

\[ SA[n]ps = \text{abs}_Z(N[n]) \]
\[ SA[x]ps = ps \times x \]
\[ SA[a_1 + a_2]ps = SA[a_1]ps +_S SA[a_2]ps \]
\[ SA[a_1 \times a_2]ps = SA[a_1]ps \times_S SA[a_2]ps \]
\[ SA[a_1 - a_2]ps = SA[a_1]ps -_S SA[a_2]ps \]
Analysis of Boolean expressions

$$SB : \text{Bexp} \rightarrow \text{PState} \rightarrow \text{TT}$$

$$SB[\text{true}]ps = \text{TT}$$
$$SB[\text{false}]ps = \text{FF}$$

$$SB[a_1 = a_2]ps = SA[a_1]ps =_S SA[a_2]ps$$

$$SB[a_1 \leq a_2]ps = SA[a_1]ps \leq_S SA[a_2]ps$$

$$SB[\neg b]ps = \neg_T (SB[b]ps)$$

$$SB[b_1 \land b_2]ps = SB[b_1]ps \land_T SB[b_2]ps$$
Properties of values

From values to properties:

\[ \text{abs}_Z : \mathbb{Z} \rightarrow \text{Sign} \]

Operations on Sign:

\[ +_S : \text{Sign} \times \text{Sign} \rightarrow \text{Sign} \]
\[ *_S : \text{Sign} \times \text{Sign} \rightarrow \text{Sign} \]
\[ -_S : \text{Sign} \times \text{Sign} \rightarrow \text{Sign} \]
\[ =_S : \text{Sign} \times \text{Sign} \rightarrow \text{TT} \]
\[ \leq_S : \text{Sign} \times \text{Sign} \rightarrow \text{TT} \]
TT: properties of truth values

Exercise: what’s the reasoning behind each and every cell?
Analysis of statements

\[ SS : \text{Stm} \rightarrow (\text{PState} \rightarrow \text{PState}) \]

\[ SS[x := a]ps = ps[x \mapsto SA[a]ps] \]

\[ SS[\langle \rangle] = \text{id} \]

\[ SS[S_1; S_2] = SS[S_2] \circ SS[S_1] \]

\[ SS[\text{if } b \text{ then } S_1 \text{ else } S_2] = \text{cond}_S(SB[b], SS[S_1], SS[S_2]) \]

\[ SS[\text{while } b \text{ do } S] = \text{FIX } H \]

where

\[ H \ h = \text{cond}_S(SB[b], h \circ SS[S], \text{id}) \]
Conditionals on properties

\[
\text{cond}_S(f, h_1, h_2) ps =
\begin{cases}
  h_1 ps & \text{if } f ps = \text{TT} \\
  h_2 ps & \text{if } f ps = \text{FF} \\
  (h_1 ps) \sqcup_{PS} (h_2 ps) & \text{if } f ps = \text{ANY} \\
  \text{INIT} & \text{if } f ps = \text{NONE}
\end{cases}
\]

INIT \( x = \text{NONE} \) for all \( x \)

Regular denotational semantics for comparison:

\[
\text{cond}(p, g_1, g_2) s =
\begin{cases}
  g_1 s & \text{if } p s = \text{tt} \\
  g_2 s & \text{if } p s = \text{ff} \\
  \text{undef} & \text{otherwise}
\end{cases}
\]
Partial order on \textit{functions} (e.g., states)

Assume that $S$ is a non-empty set and that $(D, \sqsubseteq)$ is a partially ordered set. Let $\sqsubseteq'$ be the ordering on the set $S \to D$ defined by

$$f_1 \sqsubseteq' f_2$$

if and only if

$$f_1 \ x \sqsubseteq f_2 \ x \text{ for all } x \in S$$

Then $(S \to D, \sqsubseteq')$ is a partially ordered set. Furthermore, $(S \to D, \sqsubseteq')$ is a ccppo if $D$ is and it is a complete lattice if $D$ is. In both cases we have

$$(\sqcup' Y) \ x = \sqcup \{ f \ x \mid f \in Y \}$$

so that least upper bounds are determined pointwise.
Complete lattices (again)

A partially ordered set \((D, \sqsubseteq)\) is called a *chain complete* partially ordered set (abbreviated *ccpo*) whenever \(\sqcup Y\) exists for all chains \(Y\). It is a *complete lattice* if \(\sqcup Y\) exists for all subsets \(Y\) of \(D\).

![Diagram of a complete lattice with elements T, F, ⊥, ANY, TT, FF, NONE]
Sample analysis (Factorial)

\[ SS[y := 1; \]
\[ \text{while } \neg(x \leq 1) \]
\[ \text{do } (y := y \times x; x := x - 1)] \]

\[ p_{s0} = (\text{FIX } H)(p_{s0}[y \mapsto \text{POS}]) \]

\[ H \ h = \text{cond}_S(\text{SB}[\neg(x \leq 1)], \]
\[ h \circ h_{fac}, \]
\[ \text{id}) \]

\[ h_{fac} = SS[y := y \times x; x := x - 1] \]
Fixed-point iteration: apply function to bottom ("⊥") as many times as needed to converge

Computation of iterands

for \( ps \ x = p \in \{ \text{POS, ANY} \} \)
and \( ps \ y = \text{POS} \)

\[
\begin{align*}
\text{INIT} & = \text{NONE for all } x \\
(\text{because condition is undefined}) \\
\text{So we don't even know that } y \text{ is positive for the factorial function! What's going on?}
\end{align*}
\]

\[
\begin{align*}
H^0 \perp ps & = \text{INIT} \\
H^1 \perp ps & = ps [x:=\text{Any}] \\
H^2 \perp ps & = ps [x:=\text{Any}, y:=\text{Any}]
\end{align*}
\]
Conditionals on properties

\[ \text{cond}_S(f, h_1, h_2)ps = \]
\[
\begin{cases}
  h_1 ps & \text{if } f ps = \text{TT} \\
  h_2 ps & \text{if } f ps = \text{FF} \\
  (h_1 ps) \sqcup_{PS} (h_2 ps) & \text{if } f ps = \text{ANY} \\
  \text{INIT} & \text{if } f ps = \text{NONE}
\end{cases}
\]

\text{INIT } x = \text{NONE} \text{ for all } x

Source of imprecision: we may end up with \textbf{Any} pretty quickly!
Conditionals on properties

\[
\text{FILTER}_T(f, ps) = \{ \text{ps}' \mid \text{ps}' \sqsubseteq_{PS} \text{ps}, \text{ps}' \text{ is atomic, } \text{TT} \sqsubseteq_T f \text{ ps}' \} 
\]

\[
\text{FILTER}_F(f, ps) \text{ is defined in a similar way}
\]

\[
\text{cond}_S(f, h_1, h_2)ps = \begin{cases} 
  h_1 \text{ ps} & \text{if } f \text{ ps} = \text{TT} \\
  h_2 \text{ ps} & \text{if } f \text{ ps} = \text{FF} \\
  (h_1 (\sqcup_{PS} \text{FILTER}_T(f, ps))) & \text{if } f \text{ ps} = \text{ANY} \\
  (h_2 (\sqcup_{PS} \text{FILTER}_F(f, ps))) & \text{if } f \text{ ps} = \text{ANY} \\
  \text{INIT} & \text{if } f \text{ ps} = \text{NONE}
\end{cases}
\]

These are all property states with concrete signs such that \( f \) evaluates to (not less than) TT.

(\( h_1 \text{ ps} \sqcup_{PS} h_2 \text{ ps} \) if \( f \text{ ps} = \text{ANY} \) is replaced by ... )

FYI only
The improvement

- We can do better when \( f \cdot ps = \text{ANY} \).

Key observations:

- For all states \( s \) there is a best property state \( abs(s) \) where all variables \( x \) are mapped to one of \( \text{POS} \), \( \text{ZERO} \) or \( \text{NEG} \) – such property states are called **atomic**.

- When considering the true (false) branch we can restrict attention to the atomic states that are captured by \( ps \) and where the condition could evaluate to TT (FF).
Result after improvement

For all \( n \geq 2 \)

\[
H^n \downarrow ps = ps[x \mapsto \text{ANY}]
\]

when \( ps \times \in \{\text{POS, ANY}\} \)

And it then follows that

\[
(FIX \ H)(ps_0[y \mapsto \text{POS}])
= ps_0[x \mapsto \text{ANY}][y \mapsto \text{POS}]
\]

Hence, the analysis makes a useful prediction of the sign of \( y \).
Implementation of sign detection

• Rehash denotational semantics (direct style)

• Go from standard semantics to non-standard semantics
  ✦ Define abstract domains
  ✦ Define combinators
  ✦ Migrate function signatures and equations
Standard semantics

main =
do
let s x = if x=="x" then 5 else undefined
print $ stm factorial s "y"

> main
120

https://slps.svn.sourceforge.net/svnroot/slps/topics/NielsonN07/Haskell/src/While/DenotationalSemantics/Main0.hs
Sign detection

main = do
    let xpos = update "x" Pos bottom
    print xpos
    print $ stm factorial xpos

> main
["x",Pos]
["x",TopSign],("y",TopSign)]

There is also a more precise version.

https://slps.svn.sourceforge.net/svnroot/slps/topics/NielsonN07/Haskell/src/While/SignDetection/Main0.hs
Standard semantics

-- Denotation types
  type MA = State -> Num
  type MB = State -> Bool
  type MS = State -> State

-- States
  type State = Var -> Num

-- Standard semantic functions
  aexp :: Aexp -> MA
  bexp :: Bexp -> MB
  stm :: Stm -> MS
Sign detection

-- Denotation types
type MA = PState -> Sign
type MB = PState -> TT
type MS = PState -> PState

-- Property states
type PState = Map Var Sign

-- Non-standard semantic functions
aexp :: Aexp -> MA
bexp :: Bexp -> MB
stm :: Stm -> MS
Abstract domain for truth values

data TT = BottomTT | TT | FF | TopTT

notTT :: TT -> TT
andTT :: TT -> TT -> TT

class EqTT x where (==.) :: x -> x -> TT

class OrdTT x where (<=.) :: x -> x -> TT

notTT TT     = FF
notTT FF     = TT
...

© Ralf Lämmel, 2009-2012 unless noted otherwise
Abstract domain for truth values

instance POrd TT
where
  BottomTT  <= _       = True
  _         <= TopTT   = True
  b1        <= b2      = b1 == b2

instance Bottom TT where bottom = BottomTT
instance Top TT where top = TopTT

instance Lub TT where
  b1 `lub` b2 = if b1 <= b2 then b2 else
               if b2 <= b1 then b1 else
               top
Abstract domain for numbers

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

instance Num Sign where ...
instance EqTT Sign where ...
instance OrdTT Sign where ...
instance POrd Sign where ...
instance Bottom Sign where ...
instance Top Sign where ...
instance Lub Sign where ...
instance Num Sign where

  signum = id

  abs BottomSign = BottomSign
  abs TopSign = TopSign
  abs Zero = Zero
  abs Pos = Pos
  abs Neg = Pos

  fromInteger n
      | n > 0    = Pos
      | n < 0    = Neg
      | otherwise = Zero

  ... + ... = ...
  ... * ... = ...
  ... - ... = ...

© Ralf Lämmel, 2009-2012 unless noted otherwise
Abstract domain for states

newtype (Eq k, Bottom v)
    => Map k v
    = Map { getMap :: [(k,v)] }

lookup :: (Eq k, Bottom v) => k -> Map k v -> v
lookup _ (Map []) = bottom
lookup k (Map ((k',v):m))
    = if (k == k') then v else lookup k (Map m)

update :: (Eq k, Bottom v) => k -> v -> Map k v -> Map k v
update k v m = if isBottom v then m else ...
Standard semantics

aexp :: Aexp -> MA
aexp (Num n) s   = n
aexp (Var x) s   = s x
aexp (Add a1 a2) s = aexp a1 s + aexp a2 s
aexp (Mul a1 a2) s = aexp a1 s * aexp a2 s
aexp (Sub a1 a2) s = aexp a1 s − aexp a2 s
Sign detection

\[
\text{aexp} :: \text{Aexp} \rightarrow \text{MA} \\
\text{aexp} (\text{Num } n) s = \text{fromInteger } n \\
\text{aexp} (\text{Var } x) s = \text{lookup } x s \\
\text{aexp} (\text{Add } a1 a2) s = \text{aexp } a1 s + \text{aexp } a2 s \\
\text{aexp} (\text{Mul } a1 a2) s = \text{aexp } a1 s \ast \text{aexp } a2 s \\
\text{aexp} (\text{Sub } a1 a2) s = \text{aexp } a1 s - \text{aexp } a2 s
\]
Standard semantics

bexp :: Bexp -> MB
bexp True s = Prelude.True
bexp False s = Prelude.False
bexp (Eq a1 a2) s = aexp a1 s == aexp a2 s
bexp (Leq a1 a2) s = aexp a1 s <= aexp a2 s
bexp (Not b1) s = not (bexp b1 s)
bexp (And b1 b2) s = bexp b1 s && bexp b2 s
Sign detection

\[
\begin{align*}
\text{bexp} :: \text{Bexp} & \to \text{MB} \\
bexp \ \text{True} \ s & = \ TT \\
bexp \ \text{False} \ s & = \ FF \\
bexp \ (\text{Eq} \ a1 \ a2) \ s & = \ aexp \ a1 \ s \ .==. \ aexp \ a2 \ s \\
bexp \ (\text{Leq} \ a1 \ a2) \ s & = \ aexp \ a1 \ s \ .<=. \ aexp \ a2 \ s \\
bexp \ (\text{Not} \ b1) \ s & = \ \text{notTT} \ (\text{bexp} \ b1 \ s) \\
bexp \ (\text{And} \ b1 \ b2) \ s & = \ \text{bexp} \ b1 \ s \ `\text{andTT}` \ \text{bexp} \ b2 \ s
\end{align*}
\]
Standard semantics

\[
stm :: Stm \to MS \\
stm (Assign x a) = \lambda s x' \to \text{if } x==x' \text{ then } aexp a s \text{ else } s x' \\
stm Skip = \text{id} \\
stm (Seq s1 s2) = stm s2 \cdot stm s1 \\
stm (If b s1 s2) = \text{cond} (bexp b) (stm s1) (stm s2) \\
stm (While b s) = \text{fix} (\lambda f \to \text{cond} (bexp b) (f \cdot stm s) \text{id})
\]
Sign detection

\[
\text{stm} :: \text{Stm} \rightarrow \text{MS} \\
\text{stm} (\text{Assign} \ x \ a) = \lambda s \rightarrow \text{update} \ x \ (\text{aexp} \ a \ s) \ s \\
\text{stm} \ \text{Skip} = \text{id} \\
\text{stm} (\text{Seq} \ s1 \ s2) = \text{stm} \ s2 \ . \ \text{stm} \ s1 \\
\text{stm} (\text{If} \ b \ s1 \ s2) = \text{cond} \ (\text{bexp} \ b) \ (\text{stm} \ s1) \ (\text{stm} \ s2) \\
\text{stm} (\text{While} \ b \ s) = \text{fix} \ (\lambda f \rightarrow \text{cond} \ (\text{bexp} \ b) \ (f \ . \ \text{stm} \ s) \ \text{id})
\]
Standard semantics

cond :: MB -> MS -> MS -> MS
cond b s1 s2 s = if b s then s1 s else s2 s
Sign detection

cond :: MB -> MS -> MS -> MS
cond = \mb ms1 ms2 s ->
case mb s of
  TT -> ms1 s
  FF -> ms2 s
  TopTT -> ms1 s `lub` ms2 s
  BottomTT -> bottom
Standard semantics

fix :: (x -> x) -> x
fix f = f (fix f)

fix f returns a value x such that \( fx = x \)
Sign detection

fix :: (Bottom x, Eq x) => ((x -> x) -> x -> x) -> x -> x
fix f x = iterate (const bottom)
  where
    iterate r = let r' = f r in
        if (r x == r' x)
            then r x
            else iterate r'
• **Summary**: Program analysis
  ♦ Program analyses are non-standard semantics.
    ★ Semantic domains are abstract domains.
    ★ Combinators are re-defined on abstract domains.
    ★ Semantic functions are essentially unchanged.
  ♦ Program analyses are easily expressed in Haskell.

• **Prepping**: “Semantics with applications”
  ♦ Chapter on program analysis