```
x = 1
```

```
let x = 1 in ...
```

```
x(1).
```

```
x.set(1)
```

```
!x(1)
```

Programming Language Theory

Program Analysis

Ralf Lämmel

Program analysis--what for?

- Compilation
  - ✦ Optimization
- IDE
  - ✦ Find programming errors
  - ✦ Check pre-conditions of refactorings
- 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: $\mathtt{x} := 5;\ \mathtt{y} := \mathtt{x} \star \mathtt{x} + 25$

Analysis: y evaluates to 50.

Optimized program: $\mathtt{x} := 5;\ \mathtt{y} := 50$

Example 2

Sign analysis: determine the sign of an expression.

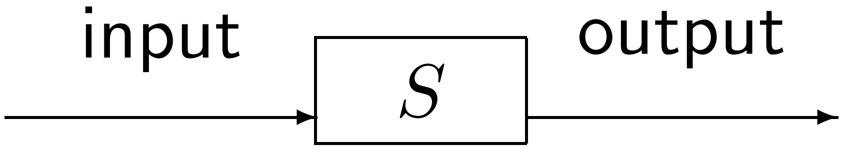
Program: $\texttt{y} := \texttt{x} \star \texttt{x} + 25;\ \texttt{while y} \leq 0\ \texttt{do} \cdots$

Analysis: y is always positive.

Optimized program: $\texttt{y} := \texttt{x} \star \texttt{x} + 25$

This slide is derived from the book & slides by Nielson & Nielson: “Semantics with applications” (1991 & 1999+).

Classes of program analysis



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

Detection of signs or constant propagation

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

Derivation of weakest pre-conditions

Program analysis and the halting problem

program analysis

$\equiv$

how to get information about programs without running them

unsolvability of the halting problem

$\Downarrow$

tell the truth

but not the complete truth

This slide is derived from the book & slides by Nielson & Nielson: “Semantics with applications” (1991 & 1999+).

Detection of Signs Analysis (Motivation)

Example

What is the sign of $(0 - 5) * 3$?

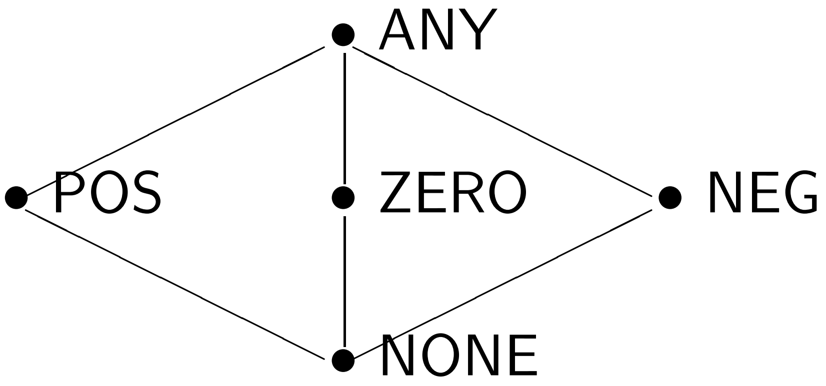
$$\underbrace{\underbrace{(\underbrace{0}_{\text{ZERO}} - \underbrace{5}_{\text{POS}})}_{\text{NEG}} * \underbrace{3}_{\text{POS}}}_{\text{NEG}}$$

Required rules for calculating with signs

$*_S$	POS	ZERO	NEG
POS	POS	ZERO	NEG
ZERO	ZERO	ZERO	ZERO
NEG	NEG	ZERO	POS

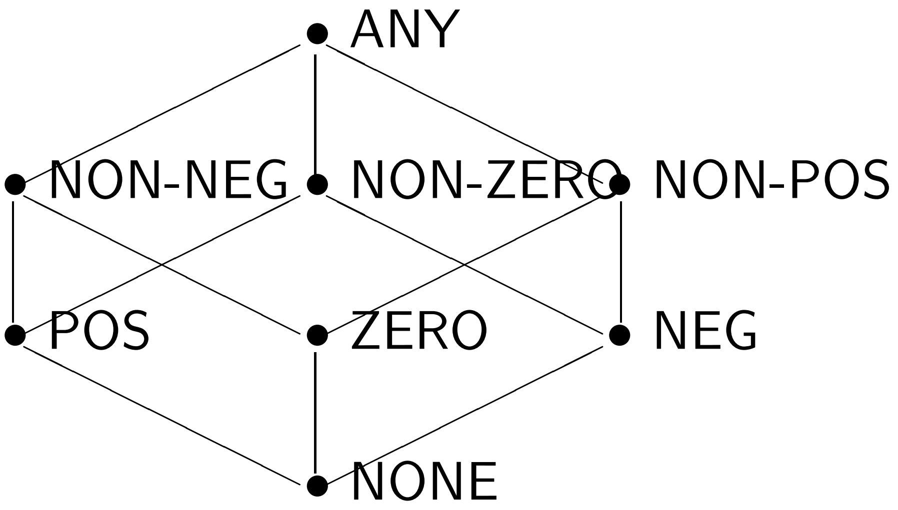
$-_S$	POS	ZERO	NEG
POS	ANY	POS	POS
ZERO	NEG	ZERO	POS
NEG	POS	NEG	ANY
ANY	ANY	ANY	ANY

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: $\mathsf{State} = \mathsf{Var} \to \mathsf{Z}$

by property states: $\mathsf{PState} = \mathsf{Var} \to \mathsf{P}_Z$

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:

- $\mathcal{A} : \text{Aexp} \to \text{State} \to \text{Z}$
- $\mathcal{B} : \text{Bexp} \to \text{State} \to \text{T}$
- $\mathcal{S}_{ds} : \text{Stm} \to (\text{State} \hookrightarrow \text{State})$

This slide is derived from the book & slides by Nielson & Nielson: "Semantics with applications" (1991 & 1999+).

From denotational semantics to program analysis

Direct style denotational semantics:

- $\mathcal{A} : \text{Aexp} \rightarrow \text{State} \rightarrow \text{Z}$
- $\mathcal{B} : \text{Bexp} \rightarrow \text{State} \rightarrow \text{T}$
- $\mathcal{S}_{ds} : \text{Stm} \rightarrow (\text{State} \hookrightarrow \text{State})$

Forward program analysis:

- $\mathcal{FA} : \text{Aexp} \rightarrow \text{PState} \rightarrow \text{P}_Z$
- $\mathcal{FB} : \text{Bexp} \rightarrow \text{PState} \rightarrow \text{P}_T$
- $\mathcal{FS} : \text{Stm} \rightarrow \text{PState} \rightarrow \text{PState}$

From denotational semantics to program analysis

Forward program analysis:

- $\mathcal{FA}$: Aexp $\rightarrow$ PState $\rightarrow$ P_Z
- $\mathcal{FB}$: Bexp $\rightarrow$ PState $\rightarrow$ P_T
- $\mathcal{FS}$: Stm $\rightarrow$ PState $\rightarrow$ PState

Backward program analysis:

- $\mathcal{BA}$: Aexp $\rightarrow$ P_Z $\rightarrow$ PState
- $\mathcal{BB}$: Bexp $\rightarrow$ P_T $\rightarrow$ PState
- $\mathcal{BS}$: Stm $\rightarrow$ PState $\rightarrow$ PState

Application of a forward analysis

Express assumptions about program variables in the beginning.

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

Requires special fixed-point approach to guarantee **termination**!

This slide is derived from the book & slides by Nielson & Nielson: "Semantics with applications" (1991 & 1999+).

Let's define a sign analysis.

Direct style denotational semantics:

$$\mathsf{State} = \mathsf{Var} \to \mathsf{Z}$$

$$\mathcal{A} : \mathsf{Aexp} \to \mathsf{State} \to \mathsf{Z}$$

$$\mathcal{B} : \mathsf{Bexp} \to \mathsf{State} \to \mathsf{T}$$

$$\mathcal{S}_{ds} : \mathsf{Stm} \to (\mathsf{State} \hookrightarrow \mathsf{State})$$

Detection of signs analysis:

$$\mathsf{PState} = \mathsf{Var} \to \mathsf{Sign}$$

$$\mathcal{SA} : \mathsf{Aexp} \to \mathsf{PState} \to \mathsf{Sign}$$

$$\mathcal{SB} : \mathsf{Bexp} \to \mathsf{PState} \to \mathsf{TT}$$

$$\mathcal{SS} : \mathsf{Stm} \to \mathsf{PState} \to \mathsf{PState}$$

Analysis of arithmetic expressions

$$\mathcal{SA} : \mathsf{Aexp} \rightarrow \mathsf{PState} \rightarrow \mathsf{Sign}$$

$$\begin{array}{lcl}
\mathcal{SA}[n]ps & = & \mathsf{abs}_Z(\mathcal{N}[n]) \\
\mathcal{SA}[x]ps & = & ps\ x \\
\mathcal{SA}[a_1 + a_2]ps & = & \mathcal{SA}[a_1]ps\ +_S\ \mathcal{SA}[a_2]ps \\
\mathcal{SA}[a_1\ *\ a_2]ps & = & \mathcal{SA}[a_1]ps\ *_S\ \mathcal{SA}[a_2]ps \\
\mathcal{SA}[a_1 - a_2]ps & = & \mathcal{SA}[a_1]ps\ -_S\ \mathcal{SA}[a_2]ps
\end{array}$$

Analysis of Boolean expressions

$$\mathcal{SB} : \mathsf{Bexp} \to \mathsf{PState} \to \mathsf{TT}$$

$$\begin{array}{lcl}
\mathcal{SB}[\texttt{true}]ps & = & \mathsf{TT} \\
\mathcal{SB}[\texttt{false}]ps & = & \mathsf{FF} \\
\mathcal{SB}[a_1 = a_2]ps & = & \mathcal{SA}[a_1]ps \ =_S \ \mathcal{SA}[a_2]ps \\
\mathcal{SB}[a_1 \leq a_2]ps & = & \mathcal{SA}[a_1]ps \ \leq_S \ \mathcal{SA}[a_2]ps \\
\mathcal{SB}[\neg b]ps & = & \neg_T \ (\mathcal{SB}[b]ps) \\
\mathcal{SB}[b_1 \ \wedge \ b_2]ps & = & \mathcal{SB}[b_1]ps \ \ \wedge_{\ T} \mathcal{SB}[b_2]ps
\end{array}$$

Properties of values

From values to properties:

abs_Z: Z → Sign

Operations on Sign:

$+_S$: Sign × Sign → Sign

$*_S$: Sign × Sign → Sign

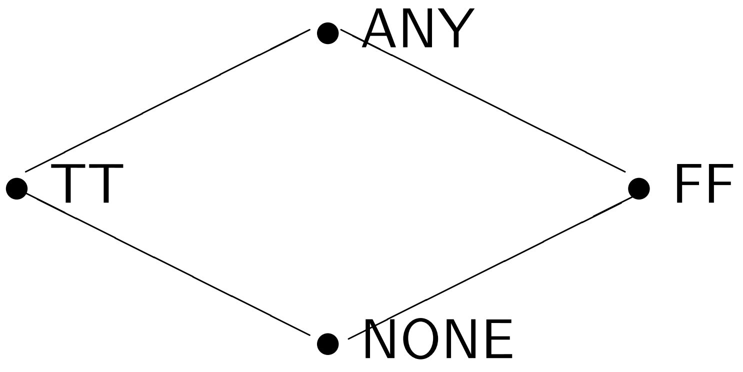
$-_S$: Sign × Sign → Sign

$=_S$: Sign × Sign → TT

$\leq_S$: Sign × Sign → TT

This slide is derived from the book & slides by Nielson & Nielson: “Semantics with applications” (1991 & 1999+).

TT: properties of truth values



$\neg_T$	NONE	TT	FF	ANY
	NONE	FF	TT	ANY

$\wedge_T$	NONE	TT	FF	ANY
NONE	NONE	NONE	NONE	NONE
TT	NONE	TT	FF	ANY
FF	NONE	FF	FF	FF
ANY	NONE	ANY	FF	ANY

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

Analysis of statements

$$\mathcal{SS} : \mathsf{Stm} \to (\mathsf{PState} \to \mathsf{PState})$$

$$\mathcal{SS}[x := a]ps = ps[x \mapsto \mathcal{SA}[a]ps]$$

$$\mathcal{SS}[\texttt{skip}] = \mathsf{id}$$

$$\mathcal{SS}[S_1; S_2] = \mathcal{SS}[S_2] \circ \mathcal{SS}[S_1]$$

$$\mathcal{SS}[\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2] = \\ \mathsf{cond}_S(\mathcal{SB}[b],\ \mathcal{SS}[S_1],\ \mathcal{SS}[S_2])$$

$$\mathcal{SS}[\texttt{while}\ b\ \texttt{do}\ S] = \mathsf{FIX}\ H$$

where

$$H\ h = \mathsf{cond}_S(\mathcal{SB}[b],\ h \circ \mathcal{SS}[S],\ \mathsf{id})$$

This slide is derived from the book & slides by Nielson & Nielson: “Semantics with applications” (1991 & 1999+).

Conditionals on properties

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

$\mathsf{INIT}\ x = \mathsf{NONE}$ for all x

Least upper bound

Regular denotational semantics for comparison:

$$\mathsf{cond}(p, g_1, g_2)\ s = \begin{cases} g_1\ s & \text{if } p\ s = \mathsf{tt} \\ & \text{and } g_1\ s \neq \mathsf{undef} \\ g_2\ s & \text{if } p\ s = \mathsf{ff} \\ & \text{and } g_2\ s \neq \mathsf{undef} \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

Partial order on ***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 ccpo if D is and it is a complete lattice if D is. In both cases we have

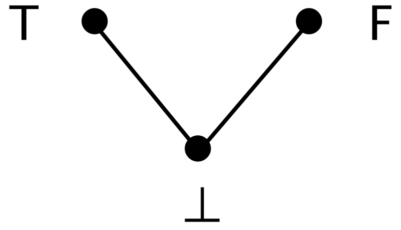
$$(\sqcup' Y)\ x = \sqcup\{f\ x \mid f \in Y\}$$

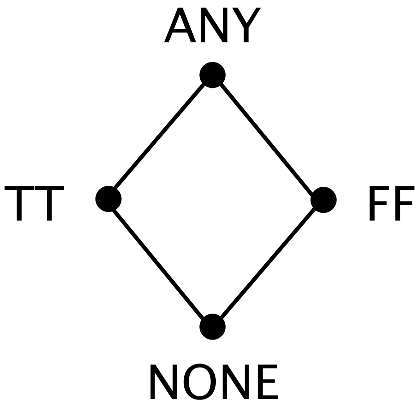
so that least upper bounds are determined pointwise.

This slide is derived from the book & slides by Nielson & Nielson: "Semantics with applications" (1991 & 1999+).

Complete lattices (again)

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





This slide is derived from the book & slides by Nielson & Nielson: "Semantics with applications" (1991 & 1999+).

Sample analysis (Factorial)

$$\mathcal{SS}[\texttt{y := 1;}$$
$$\quad\texttt{while}\ \neg\texttt{(x}\ \leq\ \texttt{1)}$$
$$\quad\texttt{do (y := y * x; x := x - 1)}]$$
$$ps_0$$
$$= (\mathsf{FIX}\ H)\ (ps_0[\mathsf{y} \mapsto \mathsf{POS}])$$

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

$$h_{fac} = \mathcal{SS}[\texttt{y := y * x; x := x - 1}]$$

Initial state

Effect of y := 1

Conditional for loop

Meaning of body

Fixed-point iteration: apply function to bottom ("⊥") as many times as needed to converge

Computation of iterands

for ps x $= p \in \{$POS, ANY$\}$

and ps y $=$ POS

INIT $x =$ NONE for all x

(because condition is undefined)

- $H^0 \perp ps =$ INIT
- $H^1 \perp ps =$ ps [x:=**Any**]
- $H^2 \perp ps =$ ps [x:=**Any**, y:=**Any**]

So we don't even know that y is positive for the factorial function! What's going on?

Conditionals on properties

$$\mathsf{cond}_S(f, h_1, h_2)ps =$$

$$\begin{cases} h_1\ ps & \text{if } f\ ps = \mathsf{TT} \\ h_2\ ps & \text{if } f\ ps = \mathsf{FF} \\ (h_1\ ps) \sqcup_{PS} (h_2\ ps) & \text{if } f\ ps = \mathsf{ANY} \\ \mathsf{INIT} & \text{if } f\ ps = \mathsf{NONE} \end{cases}$$

$\mathsf{INIT}\ x = \mathsf{NONE}$ for all x

Source of imprecision: we may end up with **Any** pretty quickly!

Conditionals on properties

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

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

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

$$\text{cond}_S(f, h_1, h_2)ps = \begin{cases} h_1\ ps \text{ if } f\ ps = \text{TT} \\ h_2\ ps \text{ if } f\ ps = \text{FF} \\ \boxed{\begin{array}{l} (h_1\ (\sqcup_{PS} \text{FILTER}_T(f, ps)) \\ \quad \sqcup_{PS}(h_2\ (\sqcup_{PS} \text{FILTER}_F(f, ps))) \\ \qquad \text{if } f\ ps = \text{ANY} \end{array}} \\ \text{INIT if } f\ ps = \text{NONE} \end{cases}$$

$(h_1\ ps) \sqcup_{PS} (h_2\ ps)$ if $f\ ps = \text{ANY}$

is replaced by ...

FYI only

The improvement

FYI only

- We can do better when *f ps* = **ANY**.

  Key observations:

  - ✦ For all states *s* there is a best property state *abs*(*s*) where all variables *x* are mapped to one of **POS**, **ZERO** or **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

FYI only

For all $n \geq 2$

$$H^n \perp ps = ps[\mathsf{x} \mapsto \mathsf{ANY}]$$

when $ps\ \mathsf{x} \in \{\mathsf{POS}, \mathsf{ANY}\}$

And it then follows that

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

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
...
```

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 ...
```

Signs as numbers

```
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

  ... + ... = ...
  ... * ... = ...
  ... – ... = ...
```

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

```
aexp :: Aexp -> MA
aexp (Num n) s = fromInteger n
aexp (Var x) s = lookup x s
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
```

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

```
bexp :: Bexp -> MB
bexp True  s            = TT
bexp False s            = FF
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      = notTT (bexp b1 s)
bexp (And b1 b2) s      = bexp b1 s `andTT` bexp b2 s
```

Standard semantics

```
stm  :: Stm  -> MS
stm (Assign x a) = \s x' -> if x==x' then aexp a s else s x'
stm Skip            = id
stm (Seq s1 s2)  = stm s2 . stm s1
stm (If b s1 s2)   = cond (bexp b) (stm s1) (stm s2)
stm (While b s)   = fix (\f -> cond (bexp b) (f . stm s) id)
```

Sign detection

```
stm  :: Stm  -> MS
stm (Assign x a) = \s -> update x (aexp a s) s
stm Skip           = id
stm (Seq s1 s2)  = stm s2 . stm s1
stm (If b s1 s2)  = cond (bexp b) (stm s1) (stm s2)
stm (While b s)   = fix (\f -> cond (bexp b) (f . stm s) 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 *f x* = *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*