EC3204: Programming Languages and Compilers

Lecture 12 — Semantic Analysis (3)
Implementation of Sign Analysis

Sunbeom So
Fall 2024

Language

The full implementation can be found at:

https://github.com/gist-pal/ec3204-pl-and-compilers/blob/main/ocaml-examples/lec12/signAnalysis.ml

$$\begin{array}{lcl} a & \rightarrow & n \mid x \mid a_1 + a_2 \mid a_1 \star a_2 \mid a_1 - a_2 \\ b & \rightarrow & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\ c & \rightarrow & x := a \mid \texttt{skip} \mid c_1; c_2 \mid \texttt{if}\ b\ c_1\ c_2 \mid \texttt{while}\ b\ c \end{array}$$

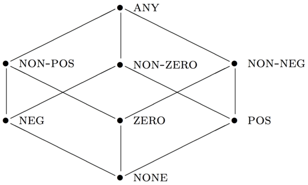
```
type aexp =
  | Int of int
  | Var of var
  | Plus of aexp * aexp
  | Mul of aexp * aexp
  | Sub of aexp * aexp

and var = string
...
```

Abstract Domain (Abstract Integers)

The abstract domain is defined as a pair $(\mathbf{Sign}, \sqsubseteq)$:

$$\mathbf{Sign} = \{\top, \bot, \text{Pos}, \text{Neg}, \text{Zero}, \text{Non-Pos}, \text{Non-Neg}, \text{Non-Zero}\}$$

where $\top$=ANY, $\bot$=NONE, and the partial order ($\sqsubseteq$) is defined as:




```
module Sign = struct
  type t = Top | Bot | Pos | Neg | Zero | NonPos | NonNeg | NonZero
  let porder : t -> t -> bool
  = fun s1 s2 ->
    match s1,s2 with
    | _ when s1 = s2 -> true | Bot,_ -> true | _,Top -> true
    | Neg,NonPos -> true | Neg,NonZero -> true | Zero,NonPos -> true
    | ...
```

Abstract Domain (Abstract Booleans)

The truth values $\mathbf{T} = \{\mathit{true}, \mathit{false}\}$ are abstracted by the complete lattice $(\widehat{\mathbf{T}}, \sqsubseteq)$:

$$\widehat{\mathbf{T}} = \{\top, \bot, \widehat{\mathit{true}}, \widehat{\mathit{false}}\}$$

$$\widehat{b_1} \sqsubseteq \widehat{b_2} \iff \widehat{b_1} = \widehat{b_2} \ \vee \ \widehat{b_1} = \bot \ \vee \ \widehat{b_2} = \top$$

```
module AbsBool = struct
  type t = Top | Bot | True | False

  let porder : t -> t -> bool
  = fun b1 b2 ->
    if b1 = b2 then true
    else
      match b1,b2 with
      | Bot,_ -> true
      | _,Top -> true
      | _ -> false
  ...
end
```

Abstract Memory State

The value abstraction is extended to the memory abstraction. The complete lattice of abstract states $(\widehat{\mathbf{State}}, \sqsubseteq)$:

$$\widehat{\mathbf{State}} = Var \to \mathbf{Sign}$$

with the pointwise ordering:

$$\hat{s}_1 \sqsubseteq \hat{s}_2 \iff \forall x \in Var.\ \hat{s}_1(x) \sqsubseteq \hat{s}_2(x).$$

```
module AbsMem = struct
  module Map = Map.Make(String) (* key domain: variable *)
  type t = Sign.t Map.t (* map domain: var -> Sign.t *)

  let porder : t -> t -> bool
  = fun m1 m2 ->
    Map.for_all (fun x v -> Sign.porder v (find x m2)) m1
  ...
end
```

Abstract Semantics for Arithmetics

$$
\begin{array}{rcl}
\widehat{\mathcal{A}}[\![a]\!] & : & \widehat{\mathsf{State}} \to \mathsf{Sign} \\
\widehat{\mathcal{A}}[\![n]\!](\hat{s}) & = & \alpha_{\mathsf{Sign}}(\{n\}) \\
\widehat{\mathcal{A}}[\![x]\!](\hat{s}) & = & \hat{s}(x) \\
\widehat{\mathcal{A}}[\![a_1 + a_2]\!](\hat{s}) & = & \widehat{\mathcal{A}}[\![a_1]\!](\hat{s}) +_{\mathsf{Sign}} \widehat{\mathcal{A}}[\![a_2]\!](\hat{s}) \\
\widehat{\mathcal{A}}[\![a_1 \star a_2]\!](\hat{s}) & = & \widehat{\mathcal{A}}[\![a_1]\!](\hat{s}) \star_{\mathsf{Sign}} \widehat{\mathcal{A}}[\![a_2]\!](\hat{s}) \\
\widehat{\mathcal{A}}[\![a_1 - a_2]\!](\hat{s}) & = & \widehat{\mathcal{A}}[\![a_1]\!](\hat{s}) -_{\mathsf{Sign}} \widehat{\mathcal{A}}[\![a_2]\!](\hat{s})
\end{array}
$$

```
let rec eval_a : aexp -> AbsMem.t -> Sign.t
= fun a m ->
  match a with
  | Int n -> Sign.alpha' n
  | Var x -> AbsMem.find x m
  | Plus (a1, a2) -> Sign.add (eval_a a1 m) (eval_a a2 m)
...
module Sign = struct
  let add : t -> t -> t
  = fun s1 s2 ->
    match s1,s2 with
    | Bot,_ | _,Bot -> Bot | Top,_ | _,Top -> Top
    | Neg,Neg -> Neg | Neg,Zero -> Neg | Neg,NonPos -> Neg ...
```

Abstract Semantics for Booleans

$$
\begin{array}{rcl}
\widehat{\mathcal{B}}[\![\, b \,]\!] & : & \widehat{\textsf{State}} \to \widehat{\textsf{T}} \\
\widehat{\mathcal{B}}[\![\, \texttt{true} \,]\!](\hat{s}) & = & \widehat{true} \\
\widehat{\mathcal{B}}[\![\, \texttt{false} \,]\!](\hat{s}) & = & \widehat{false} \\
\widehat{\mathcal{B}}[\![\, a_1 = a_2 \,]\!](\hat{s}) & = & \widehat{\mathcal{A}}[\![\, a_1 \,]\!](\hat{s}) =_{\textsf{Sign}} \widehat{\mathcal{A}}[\![\, a_2 \,]\!](\hat{s}) \\
\widehat{\mathcal{B}}[\![\, a_1 \leq a_2 \,]\!](\hat{s}) & = & \widehat{\mathcal{A}}[\![\, a_1 \,]\!](\hat{s}) \leq_{\textsf{Sign}} \widehat{\mathcal{A}}[\![\, a_2 \,]\!](\hat{s}) \\
\widehat{\mathcal{B}}[\![\, \neg b \,]\!](\hat{s}) & = & \neg_{\widehat{\textsf{T}}} \widehat{\mathcal{B}}[\![\, b \,]\!](\hat{s}) \\
\widehat{\mathcal{B}}[\![\, b_1 \land b_2 \,]\!](\hat{s}) & = & \widehat{\mathcal{B}}[\![\, b_1 \,]\!](\hat{s}) \land_{\widehat{\textsf{T}}} \widehat{\mathcal{B}}[\![\, b_2 \,]\!](\hat{s})
\end{array}
$$

```
let rec eval_b : bexp -> AbsMem.t -> AbsBool.t
= fun b m ->
  match b with
  | True -> AbsBool.True | False -> AbsBool.False
  | Eq (a1, a2) -> Sign.eq (eval_a a1 m) (eval_a a2 m)
  | Leq (a1, a2) -> Sign.leq (eval_a a1 m) (eval_a a2 m)
  | Not b -> AbsBool.not (eval_b b m)
  | And (b1, b2) -> AbsBool.band (eval_b b1 m) (eval_b b2 m)
...
module Sign = struct ... let eq = ... end
module AbsBool = struct ... let not = ... end
```

Abstract Semantics for Commands

$$
\begin{array}{rcl}
\widehat{\mathcal{C}}[\![c]\!] & : & \widehat{\textsf{State}} \to \widehat{\textsf{State}} \\
\widehat{\mathcal{C}}[\![x := a]\!] & = & \lambda\hat{s}.\hat{s}[x \mapsto \widehat{\mathcal{A}}[\![a]\!](\hat{s})] \\
\widehat{\mathcal{C}}[\![\texttt{skip}]\!] & = & \textbf{id} \\
\widehat{\mathcal{C}}[\![c_1; c_2]\!] & = & \widehat{\mathcal{C}}[\![c_2]\!] \circ \widehat{\mathcal{C}}[\![c_1]\!] \\
\widehat{\mathcal{C}}[\![\texttt{if}\ b\ c_1\ c_2]\!] & = & \widehat{\textsf{cond}}(\widehat{\mathcal{B}}[\![b]\!], \widehat{\mathcal{C}}[\![c_1]\!], \widehat{\mathcal{C}}[\![c_2]\!]) \\
\widehat{\mathcal{C}}[\![\texttt{while}\ b\ c]\!] & = & \lambda\hat{s}.\mathit{fix}(\lambda\hat{x}.\hat{s} \sqcup \widehat{\mathcal{C}}[\![c]\!](\hat{x}))
\end{array}
$$

$$
\widehat{\textsf{cond}}(f, g, h)(\hat{s}) = \begin{cases} \bot & \cdots f(\hat{s}) = \bot \\ g(\hat{s}) & \cdots f(\hat{s}) = \widehat{\mathit{true}} \\ h(\hat{s}) & \cdots f(\hat{s}) = \widehat{\mathit{false}} \\ g(\hat{s}) \sqcup h(\hat{s}) & \cdots f(\hat{s}) = \top \end{cases}
$$

Abstract Semantics for Commands

- The implementation of the abstract semantics for the while-loop.
- It aims to compute "stable" abstract memory states at the **loop entry**.

```
let rec eval_c : cmd -> AbsMem.t -> AbsMem.t
= fun c m ->
   match c with
   | Assign (x,a) -> AbsMem.add x (eval_a a m) m
   | Skip -> m
   | Seq (c1,c2) -> eval_c c2 (eval_c c1 m)
   | If (b, c1, c2) -> cond (eval_b b, eval_c c1, eval_c c2) m
   | While (b, c) ->
     let onestep x = AbsMem.join m (eval_c c x) in
     let rec fix f x i =
       let x' = f x in
       if AbsMem.porder x' x then x
       else fix f x' (i+1)
     in
     fix onestep m 1
```