Homework 5

SNU 4541.664A

Due: 06/03, 09:30(design), 24:00(program)

Kwangkeun Yi

이 숙제의 목표는

- 앞으로 구현할 분석기들이 사용할 다양한 요약도메인(abstract domain)들을 자동으로 구현해 줄 모듈함수(functor)들을 정의한다.
- 프로그램의 모듬 의미(collecting semantics)와 요약 의미(abstract semantics)를 기계상태의 변환(transition semantics) 스타일로 정의해서 분석기를 디자인하고 증명해 본다.
- 디자인된 분석기를 구현하고 돌려본다.

Exercise 1 "요약 도메인 자동구현기"

프로그램 분석기에서 사용할 요약 도메인(abstract domain, cpo)들을 구현해 줄 모듈함수(functor)들을 구현한다.

예를들어 다음과 같이 정의된 도메인 $\hat{D}_i$들은

$$\begin{array}{rcll} A & = & \{a,b,c\} & \text{set} \\ B & = & \{0,1,2,3\} & \text{set} \\ C & = & A \xrightarrow{\text{fin}} B & \text{set} \\ \hat{D}_1 & = & C \cup \{\bot, \top\} & \text{lifted, flat domain} \\ \hat{D}_2 & = & 2^B & \text{powerset domain} \\ \hat{D}_3 & = & \hat{D}_1 \times \hat{D}_2 & \text{product domain} \\ \hat{D}_4 & = & A \to \hat{D}_3 & \text{atomic function domain} \end{array}$$

다음과 같이 손쉽게 구현될 것이다:

```
module A = PrimitiveSet(struct type t = string
                            let compare = compare
                            exception TooMany
                            val all = fun () -> ["a"; "b"; "c"]
                        end)

module B = PrimitiveSet(struct type t = int
                            let compare = compare
                            exception TooMany
                            val all = fun () -> [0; 1; 2; 3]
                        end)
module C = FunctionSet (A) (B)
module D1 = FlatDomain (C)
module D2 = PowersetDomain (B)
module D3 = ProductDomain (D1) (D2)
module D4 = FunDomain (A) (D3)
```

요약도메인 모듈들을 만들어 주는 모듈함수들을 구현하라. 아래의 코드에서 빈칸("...")을 완성하면 된다. (요약도메인은 집합으로 부터 만들어 질 것이므로, 집합 모듈들을 만들어주는 모듈함수(functor)들은 아래와 같이 제공된다.)

주의: 모듈 함수 `FunDomain (A) (B)`에 의해서 만들어 지는 $A \to B$는 "atomic function domain"이라고 불리는데, A는 유한집합이어야 하고 B는 cpo로서 다음의 성질을 가지는 cpo이다:

$$A \to B \quad \equiv \quad \underbrace{B \times \cdots \times B}_{|A|}$$

즉, $A \to B$의 원소들은 A의 원소들이 키가 되는 B 테이블이다.

```
(* set functors for
 * SNU 4541.664A Program Analysis
 * Kwangkeun Yi
 *)
module type SET = sig
                   include Set.S
                   exception TooMany
                   val all: unit -> t
                  end

module PrimitiveSet (A: sig
                            type t val compare: t -> t -> int
                            exception TooMany
                            val all: unit -> t list
                          end) =
 struct
  include Set.Make (A)
  exception TooMany
  let all = fun () -> try
                        List.fold_left (fun s x -> add x s) empty (A.all())
                      with A.TooMany -> raise TooMany
 end

module ProductSet (A: SET) (B: SET) =
 struct
  include Set.Make (struct type t = A.elt * B.elt let compare = compare end)
  exception TooMany
  let all = fun () -> try
                        A.fold (fun a c ->
                                          B.fold (fun b c -> add (a,b) c)
                                                 (B.all()) c
                                        ) (A.all()) empty
                       with A.TooMany -> raise TooMany
                          | B.TooMany -> raise TooMany
 end
```

```
module PowerSet (A: SET) =
 struct
  include Set.Make (struct type t = A.t let compare = compare end)
  exception TooMany
  let all = fun () -> raise TooMany
 end

module FunctionSet (A: SET) (B: SET) =
 struct
  module F = Map.Make (struct type t = A.elt let compare = compare end)
  include Set.Make (struct type t = B.elt F.t let compare = compare end)
  exception TooMany
  let all = fun () -> raise TooMany
  let domain_all = A.all
  let range_all = B.all
 end
```

```
(* domain functors for
 * SNU 4541.664A Program Analysis
 * Kwangkeun Yi
 *)

module type DOMAIN =
 sig
   type elt      (* the type of abstract domain elements *)
   val top: elt
   val bot: elt
   val join: elt -> elt -> elt
   val leq: elt -> elt -> bool
 end

module type FLAT_DOMAIN =
 sig
   include DOMAIN
   type atom
   val make: atom -> elt
 end

module type PRODUCT_DOMAIN =
 sig
  include DOMAIN
  type lelt
  type relt
  val l: elt -> lelt     (* left *)
  val r: elt -> relt     (* right *)
  val make: lelt -> relt -> elt
 end
```

```
module type POWERSET_DOMAIN =
 sig
  include DOMAIN
  type atom
  val union: elt -> elt -> elt
  val inter: elt -> elt -> elt
  val diff: elt -> elt -> elt
  val remove: atom -> elt -> elt
  val mem: atom -> elt -> bool
  val map: (atom -> atom) -> elt -> elt
  val fold: (atom -> 'a -> 'a) -> elt -> 'a -> 'a
  val make: atom list -> elt
 end

module type FUNCTION_DOMAIN =
 sig
  include DOMAIN
  type lelt
  type relt
  val image: elt -> lelt -> relt
  val update: elt -> lelt -> relt -> elt
  val map: (lelt -> relt -> lelt * relt) -> elt -> elt
  val fold: (lelt -> relt -> 'a -> 'a) -> elt -> 'a -> 'a
  val make: (lelt * relt) list -> elt
 end

module type INTERVAL_DOMAIN =
 sig
  include DOMAIN
  exception Undefined
  type bound = Z of int | Pinfty | Ninfty
  val l: elt -> bound  (* lower bound *)
  val u: elt -> bound  (* upper bound *)
  val make: bound -> bound -> elt
 end
```

```
module FlatDomain (A: SET) : FLAT_DOMAIN
 with type atom = A.elt =
 struct
  type elt = BOT | TOP | ELT of A.elt
  type atom = A.elt
  let bot = BOT
  let top = TOP
  let join x y = match (x, y)
         with (BOT, _) -> y
            | (_, BOT) -> x
            | (TOP, _) -> TOP
            | (_, TOP) -> TOP
            | (x, y) -> if x=y then x else TOP
  let leq x y = match (x, y)
        with (BOT, _) -> true
           | (_, TOP) -> true
           | (ELT a, ELT b) -> a=b
           | _ -> false
  let make a = ELT a
 end
```

```
module ProductDomain (A: DOMAIN) (B: DOMAIN) : PRODUCT_DOMAIN
 with type lelt = A.elt and type relt = B.elt =
 struct
  type elt = BOT | TOP | ELT of A.elt * B.elt
  type lelt = A.elt
  type relt = B.elt
  let bot = BOT
  let top = TOP
  let join x y = match (x,y)
                 with (BOT,_) -> y
                    | (TOP,_) -> TOP
                    | (_,BOT) -> x
                    | (_,TOP) -> TOP
                    | (ELT(a,b), ELT(a',b')) -> ELT(A.join a a', B.join b b')
  let leq x y = match (x,y)
                with (BOT,_) -> true
                   | (TOP,_) -> false
                   | (_,BOT) -> false
                   | (_,TOP) -> true
                   | (ELT(a,b), ELT(a',b')) -> (A.leq a a') && (B.leq b b')
  let l x = match x with TOP -> A.top | BOT -> A.bot | ELT(a,b) -> a
  let r x = match x with TOP -> B.top | BOT -> B.bot | ELT(a,b) -> b
  let make a b = ELT (a,b)
 end
```

```
module PowersetDomain (A: SET) : POWERSET_DOMAIN
 with type atom = A.elt =
 struct
  type elt = BOT | TOP | ELT of A.t
  type atom = A.elt
  let bot = BOT
  let top = TOP
  let join x y = match (x,y)
                 with (BOT, _) -> y
                    | (_, BOT) -> x
                    | (TOP, _) -> TOP
                    | (_, TOP) -> TOP
                    | (ELT s, ELT s') -> ELT (A.union s s')
  let mem a s = match s with BOT -> false
                           | TOP -> true
                           | ELT s -> A.mem a s
  let fold f x a = match x with BOT -> a
                              | TOP -> A.fold f (A.all()) a
                              | ELT s -> A.fold f s a
  let map f x = match x with BOT -> BOT
                           | TOP ->
                     (ELT (A.fold (fun a s' -> A.add (f a) s') (A.all()) A.empty))
                           | ELT s ->
                     (ELT (A.fold (fun a s' -> A.add (f a) s') s A.empty))

  let make lst = match lst
                  with [] -> BOT
                     | l -> ELT
                             (List.fold_left (fun s x -> A.add x s)
                                             A.empty l
                             )
  ...
 end
```

```
module FunDomain (A: SET) (B: DOMAIN) : FUNCTION_DOMAIN
 with type lelt = A.elt and type relt = B.elt =
 struct
  module Map = Map.Make(struct type t = A.elt let compare = compare end)
  type elt = BOT | TOP | ELT of B.elt Map.t
  type lelt = A.elt
  type relt = B.elt
  let bot = BOT
  let top = TOP
  ...
 end
```

□

Exercise 2 "`mod11trace`: 분석기 디자인과 구현"

이번에는 아래의 언어에 대한 분석기를 기계상태의 변환(transition semantics) 스타일로 디자인하고 구현한다. 변수의 주소를 값으로 다룰 수 있고, 참거짓 값이 따로 있는 언어 K를 생각하자:

$$
\begin{array}{rcl}
C & \to & \texttt{skip} \\
& | & x \texttt{ := } E \mid \texttt{*}x \texttt{ := } E \\
& | & C \texttt{ ; } C \\
& | & \texttt{if } E\ C\ C \\
& | & \texttt{while } E\ C \\
E & \to & n \quad (n \in \mathbb{Z}) \mid \texttt{true} \mid \texttt{false} \\
& | & x \mid \texttt{*}x \mid \texttt{\&}x \\
& | & E \texttt{ + } E \mid \texttt{ - } E \mid E \texttt{ < } E
\end{array}
$$

언어 K의 프로그램 구조를 구현한 모듈 K는 다음의 모듈타입 K를 만족하도록 한다:

```
module type K =
sig
  exception Error of string
  type id = string
  type exp = NUM of int
           | TRUE | FALSE
           | VAR of id
           | STAR of id                    (* *x *)
           | LOC of id                     (* &x *)
           | ADD of exp * exp
           | MINUS of exp
           | LESS of exp * exp

  type cmd = SKIP
           | ASSIGN of id * exp            (* assign to variable *)
           | IASSIGN of id * exp           (* indirect assign *)
           | SEQ of cmd * cmd              (* sequence *)
           | IF of exp * cmd * cmd         (* if-then-else *)
           | WHILE of exp * cmd            (* while loop *)
end
```

분석기에 입력으로 들어오는 프로그램은 빈 메모리를 받아 문제 없이 실행되는 K 프로그램이라고 가정한다. `if`와 `while`-문의 조건식은 항상 부울값(참 거짓)만 계산해야 문제없이 실행된다.

분석하고자 하는 성질: 변수들이 가지는 정수를 11로 나누었을 때 나머지의 범위이다.

제출할 것은 디자인 리포트(PDF)와 구현된 코드이다

- 디자인: 분석기에 해당하는 "요약 의미구조"(abstract semantics)를 정의하고 그 안전성을 증명한 것을 리포트로 제출한다.

- 구현: 분석기 프로그램

  $$\texttt{mod11trace} : \texttt{K.cmd} \to \texttt{unit}$$

  을 구현해서 제출한다. Exercise 1에서 정의한 모듈함수를 이용해서 요약도메인을 구현한다. 분석기 `mod11trace`는 주어진 프로그램을 분석한 후 그 결과를 출력한다. 분석결과의 출력은: 프로그램을 "depth-first-traversal"하면서 만나는 명령문 순서로, 분석된 결과(메모리 상태)를 출력하도록 한다.

분석기는 다음의 스타일로 디자인한다:

- 프로그램 C의 모듬의미(collecting semantics) $[\![C]\!]$를 C가 실행중에 만드는 모든 기계상태들의 집합으로 정의한다:

  $$[\![C]\!] \in 2^{State}$$

  물론, 다음 함수 $F \in 2^{State} \to 2^{State}$의 고정점으로 정의된다:

  $$\mathit{fix}(F \stackrel{\text{let}}{=} \lambda T. T_0 \cup \mathit{Next}\, T).$$

  위에서 정의되지 않은 모든 것들을 정의하라: 기계상태 집합 $State$,초기의 기계상태 집합 T_0, 기계상태들을 받아 다음 기계상태들을 내 놓는 함수 $\mathit{Next} \in 2^{State} \to 2^{State}$.

- 프로그램 C의 정적분석(요약해석) 결과 $[\![\hat{C}]\!]$은 하나 이상의 요약된 기계상태들의 집합으로 정의한다:

  $$[\![\hat{C}]\!] \in 2^{\hat{State}}$$

물론, 다음 함수 $\hat{F} \in 2^{\hat{State}} \to 2^{\hat{State}}$의 고정점으로 정의된다:

$$fix(\hat{F} \stackrel{\text{let}}{=} \lambda \hat{T}.\alpha(T_0) \sqcup \hat{Next}\,\hat{T})$$

위에서 정의되지 않은 모든 것들을 정의하라: 요약된 기계상태 집합(cpo) $\hat{State}$, 2^{State}와 $\hat{State}$가 안전한 요약관계

$$2^{State} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{State}$$

가 되도록 하는 갈로아 짝 α와 γ, 요약된 기계상태들을 받아 다음 상태들을 내 놓는 함수 $\hat{Next}$.

- "하나 이상의 요약된 기계상태들의 집합"$\in 2^{\hat{State}}$ 이란, 모듬의미(collecting semantics)에서 모아지는 모든 기계상태들의 집합을 여러집합으로 분할(partitioning)하고 분할된 각 집합을 따로따로 하나의 기계상태로 요약한 결과를 말한다. 이 때, 분할의 기준은 기계상태가 프로그램의 어느 지점에서 발생한 것이냐이다.

이제 해야 하는 것은:

- 위 정의들의 안전성 증명. 수업시간에 강의한 내용을 기초로, 증명이 필요한 두 개의 성질을 확인하면 된다.
  - $\hat{Next}$에서 사용되는 함수

    $$\hat{next} \in \hat{State} \to 2^{\hat{State}}$$

    의 조건:

    $$\forall s \in State : next\ s \in (\wp_\cup(\gamma) \circ \hat{next} \circ \alpha)\ \{s\}$$

    이때 $next \in State \to State$는 $Next$에서 사용된 함수.
  - 요약 분할함수 $\hat{\pi} \in 2^{\hat{State}} \to 2^{2^{\hat{State}}}$의 조건:

    $$\wp(\alpha) \circ \pi \circ \wp_\cup(\gamma) \sqsubseteq \wp(\sqcup) \circ \hat{\pi}.$$

- 위의 정의에 준한 자동 분석기의 구현.

□