Un interprete astratto per l'inferenza dei tipi

Contenuti

- l'inferenza dei tipi come interprete astratto denotazionale
- il frammento senza `Let` e l'adattamento della semantica denotazionale
- semantica concreta collecting
- verso la semantica astratta
  - il dominio dei tipi
  - perché servono i vincoli
  - il dominio astratto
- esempi di operazioni astratte
- il calcolo di punto fisso astratto ed il widening

L'inferenza dei tipi come interprete astratto denotazionale

- perché basta la semantica denotazionale
  - i tipi sono astrazioni dei valori e possono essere osservati anche sulla semantica denotazionale
    - in altre analisi (per esempio, relative al dimensionamento dei frames locali) ho bisogno di semantiche più dettagliate
  - in quasi tutte le analisi statiche, le funzioni devono essere analizzate al momento della definizione
    - mi serve una semantica vera della astrazione
    - gratis in una semantica denotazionale
    - in un interprete dettagliato può essere opportuno introdurre un trattamento "denotazionale" delle funzioni

Anche l'inferenza dei tipi comporta approssimazioni

- in ML la semantica concreta è definita solo per programmi che superano la verifica dei tipi
- la seguente espressione non può essere tipata, perché l'algoritmo di inferenza di tipi richiede che le due espressioni del condizionale abbiano lo stesso tipo

```
let x = true in if x then true else 1
```

- in realtà la semantica concreta (con controllo dei tipi "a run time") dell'espressione darebbe il valore `true`
  - l'inferenza dei tipi dovrebbe dare `bool` invece che `type error`
    - `bool` ≤ `type error`

Il frammento di linguaggio funzionale senza `Let`

```
type ide = string
type exp =  Eint of int
        | Ebool of bool
        | Den of ide
        | Prod of exp * exp
        | Sum of exp * exp
        | Diff of exp * exp
        | Eq of exp * exp
        | Minus of exp
        | Iszero of exp
        | Or of exp * exp
        | And of exp * exp
        | Not of exp
        | Ifthenelse of exp * exp * exp
        | Fun of ide  * exp
        | Appl of exp * exp
        | Rec of ide * exp
```

- con il `Let` per ottenere il comportamento di ML bisognerebbe adottare un sistema di tipi polimorfo
- funzioni con un solo argomento per semplicità

La semantica denotazionale modificata

- la separazione delle operazioni primitive da `sem` permetterà di rimpiazzarle con le versioni astratte
- nei casi di errore (di tipo) rimpiazziamo le eccezioni con il ritorno del valore indefinito `Unbound`
- dove ci sono valori concreti, introduciamo una funzione che li astrae
  - la funzione identità nella semantica concreta

```
let alfa x = x
```

- semantica "non deterministica" del condizionale

```
let valid x = match x with Bool g -> g
let unsatisfiable x = match x with Bool g -> not g
let merge (a,b,c) = b
```

Il dominio concreto

- ricordiamo la struttura del dominio concreto `eval`

```
type eval =  Int of int    | Bool of bool   | Unbound
       | Funval of efun
and efun = eval -> eval
```

- una operazione primitiva

```
let plus (x,y) = match (x,y) with
      |(Int nx, Int ny) -> Int(nx + ny)
      | _ -> Unbound
```

La semantica denotazionale modificata 1

```
let rec sem (e:exp) (r:eval env) =
      match e with
      | Eint(n) -> alfa(Int(n))
      | Ebool(b) -> alfa(Bool(b))
      | Den(i) -> applyenv(r,i)
      | Iszero(a) -> iszero((sem a r) )
      | Eq(a,b) -> equ((sem a r) ,(sem b r) )
      | Prod(a,b) ->  mult((sem a r), (sem b r))
      | Sum(a,b) ->  plus((sem a r), (sem b r))
      | Diff(a,b)  ->  diff((sem a r), (sem b r))
      | Minus(a) ->  minus((sem a r))
      | And(a,b) ->  et((sem a r), (sem b r))
      | Or(a,b) ->  vel((sem a r), (sem b r))
      | Not(a) -> non((sem a r))
      | Ifthenelse(a,b,c) -> let g = sem a r in
            if typecheck("bool",g) then
                    (if valid(g) then sem b r else
                     (if unsatisfiable(g) then sem c r
                     else merge(g, sem b r, sem c r)))
            else failwith ("nonboolean guard")
      | Fun(i,a) ->  makefun(Fun(i,a), r)
      | Appl(a,b) -> applyfun(sem a r, sem b r)
      | Rec(i,a) ->  makefunrec(i,a, r)
```

La semantica denotazionale modificata 2

```
and makefun ((a:exp),(x:eval env)) =
      (match a with
      | Fun(ii,aa) ->
         Funval(function d ->  sem aa (bind (x, ii, d)))
      | _ -> Unbound )

and applyfun ((ev1:eval),(ev2:eval)) =
      ( match ev1 with
      | Funval(x) -> x ev2
      | _ -> Unbound)

and makefunrec (i, Fun(ii, aa), r) =
    let functional ff d =
        let r1 = bind(bind(r, ii, d), i, Funval(ff)) in
              sem aa r1  in
           let rec fix =  function x -> functional fix x
               in Funval(fix)
```

Dalla semantica concreta a quella collecting

- la funzione di valutazione semantica concreta
  - `sem: exp -> eval env -> eval`
- la funzione di valutazione semantica collecting
  - ha come codominio $\mathcal{P}$(`eval`)
    - che possiamo "rappresentare" come `eval list`
  - `semc: exp -> eval env -> eval list`
  - `semc e r = [sem e r]`
  - tutti gli operatori primitivi concreti devono essere pensati come definiti su $\mathcal{P}$(`eval`)
    - per la progettazione delle loro versioni astratte
  - esistono altre semantiche collecting (più concrete)
    - `exp ->` $\mathcal{P}$(`eval env -> eval`)

Dalla semantica collecting a quella astratta

- dominio concreto: $(\mathcal{P}(\texttt{ceval}), \subseteq)$
- ambiente concreto (non-collecting):
  `ide -> ceval`
- dominio astratto: $(\texttt{eval}, \leq)$
- ambiente astratto:
  `ide -> eval`
- la funzione di valutazione semantica collecting
  `semc: exp -> ceval env -> P(ceval)`
- la funzione di valutazione semantica astratta
  `sem: exp -> eval env -> eval`

L'interprete astratto sui tipi

- corrisponde al sistema di monotipi à la Hindley
  - coincide con quello di ML in assenza di `Let`
- inferisce il tipo principale
  - monotipo con variabili
  - che sussume tutti gli altri tipi
  - rappresentato come un termine
    - con opportuni costruttori e con variabili

I monotipi con variabili

```
type evalt = Notype
    | Vvar of string
    | Intero
    | Booleano
    | Mkarrow of evalt * evalt
```

- l'ordinamento parziale (su classi di equivalenza di termini modulo varianza)
  - la relazione di anti-istanza:
    - $t_1 \leq t_2$, se t_2 è una istanza di t_1
    - Notype è l'elemento massimo
  - ci sono catene infinite crescenti (il dominio non è noetheriano)
  - ci possono essere problemi di terminazione nel calcolo del punto fisso

Verso il dominio astratto

```
type evalt = Notype
    | Vvar of string
    | Intero
    | Booleano
    | Mkarrow of evalt * evalt
```

- l'ordinamento parziale
  - $t_1 \leq t_2$, se t_2 è una istanza di t_1
  - Notype è l'elemento massimo
- *glb* di `evalt`:
  - lcg (minima comune generalizzazione), calcolata con l'algoritmo di anti-unificazione
- *lub* di `evalt`:
  - gci (massima istanza comune), calcolata con l'algoritmo di unificazione
- anche se evalt non è il vero dominio astratto, lo mettiamo in relazione con il dominio concreto

Concretizzazione

- dominio concreto: $(\mathcal{P}(\texttt{ceval}), \subseteq, \emptyset, \texttt{ceval}, \cup, \cap)$
- dominio astratto: $(\texttt{evalt}, \leq, \texttt{Vvar(\_)}, \texttt{Notype}, \text{gci}, \text{lcg})$

$\gamma_t(x) =$

$\texttt{ceval}$, se $x = \texttt{Notype}$

$\{ y \mid \exists z.\ y = \texttt{Int}(z)\}$, se $x = \texttt{Intero}$

$\{ y \mid \exists z.\ y = \texttt{Bool}(z)\}$, se $x = \texttt{Booleano}$

$\emptyset$, se $x = \texttt{Vvar(\_)}$

$\{\text{Funval}(f) \mid \forall d \in \gamma_t(\sigma)\ f(d) \in \gamma_t(\tau)\}$,

se $x = \texttt{Mkarrow}(\sigma, \tau)$, con σ, τ senza variabili

$\cap\, \gamma_t(\mu)$, per μ istanza senza variabili di x,

se $x = \texttt{Mkarrow}(\sigma, \tau)$, con variabili in σ o in τ

- semplice da definire la funzione di astrazione

L'astrazione delle funzioni

- data l'operazione concreta (non-collecting)

```
let rec makefun (Fun(ii,aa),(x:eval env)) =
   Funval(function d ->  sem aa (bind (x, ii, d)))
```

- nella versione astratta si dovrebbe
  - per ogni tipo ground τ_i
    - dare a `d` il valore τ_i
    - calcolare il tipo σ_i = `sem aa (bind(r,ii,d))`
  - calcolare il *glb* di tutti i tipi funzionali risultanti:
    - lcg ({`Mkarrow`(τ_i, σ_i)})
- può essere reso effettivo facendo una sola valutazione della semantica (astratta) del corpo, dando a `d` come valore una nuova variabile di tipo (elemento minimo)
  - operazione astratta (sbagliata!)

```
let rec makefun (Fun(ii,aa),(x:eval env)) = let d = newvar() in
    let t = sem aa (bind (x, ii, d))) in Mkarrow(d, t)
```

Le variabili di tipo tipo

- γ_t(`Vvar(_)`) = $\emptyset$
  - una variabile di tipo tipo rappresenta l'insieme di tutti i valori (concreti) che hanno tutti i tipi, cioè l'insieme vuoto
- (nuove) variabili vengono introdotte nella versione astratta di `makefun`

```
let rec makefun (Fun(ii,aa),(x:eval env)) = let d = newvar() in
    let t = sem aa (bind(x, ii, d))) in Mkarrow(d, t)
```

- il problema
  - la definizione è sbagliata perché la variabile in `d` può essere istanziata (sottoposta a vincoli) durante la valutazione del corpo della funzione `aa`

Abbiamo bisogno dei vincoli

```
let rec makefun (Fun(ii,aa),(r:eval env)) = let d = newvar() in
 let t = sem aa (bind(r, ii, d))) in Mkarrow(d, t)
```

- `Fun("x", Sum(Den("x"), Eint 1))`
  - `"x"` viene legato ad una nuova variabile `Vvar "0"` nell'ambiente `r` producendo `r1`
  - l'espressione `Sum(Den("x"), Eint 1))` è valutata in `r1`
  - la versione astratta dell'operatore `plus` deve istanziare la variabile `Vvar "0"` a `Intero`
    - astrazione del type checking concreto
- questo risultato può essere ottenuto
  - estendendo il dominio astratto a coppie consistenti di
    - un termine (monotipo)
    - un vincolo sulle variabili di tipo tipo
      - insieme di equazioni fra termini in forma risolta, come quelle costruite dall'algoritmo di unificazione

Il vero dominio astratto

```
type evalt = Notype
    | Vvar of string
    | Intero
    | Booleano
    | Mkarrow of evalt list * evalt
type eval = evalt * (evalt * evalt) list
```

- il secondo componente di ogni valore astratto (il vincolo) rappresenta un insieme di uguaglianze fra variabili e termini (equazioni in forma risolta)
- ogni operatore astratto
  - combina i vincoli degli argomenti ed aggiorna il risultato con nuovi vincoli
  - controlla che il vincolo risultante sia soddisfacibile e lo trasforma in forma risolta (attraverso l'unificazione)
  - applica il vincolo in forma risolta (sostituzione) al tipo
  - ritorna la coppia (tipo, vincolo)
- l'ordinamento, le operazioni di *lub* e *glb* e la funzione di concretizzazione per `eval` si ottengono facilmente da quelle definite per `evalt`

Termini, Equazioni, Unificazione

- immaginiamo di avere una implementazione dell'unificazione sulla particolare classe di termini `evalt`
- introduciamo il tipo sostituzione (restituito dalle varie versioni dell'unificazione)

```
type subst = Fail
           | Subst of  (evalt * evalt) list
```

- supponiamo di avere le operazioni

```
val unifylist : (evalt * evalt) list -> subst
val applysubst : subst -> evalt -> evalt
```

- supponiamo di avere anche le operazioni

```
val newvar : unit -> evalt
val abstreq : eval * eval -> bool
```

Un tipico operatore astratto

```
let plus ((v1,c1),(v2,c2))  =
  let sigma =
    unifylist((v1,Intero) :: (v2,Intero) :: (c1 @ c2)) in
  match sigma with
      |Fail -> (Notype,[])
      |Subst(s) -> (Intero,s)
```

Astrazione e applicazione

```
let rec makefun(Fun(ii,aa),r) =
  let f1 =newvar() in let f2 =newvar() in
  let body = sem aa (bind(r,ii,(f1,[]))) in
   (match body with (t,c) ->
   let sigma = unifylist( (t,f2) :: c) in
    (match sigma with
     |Fail -> (Notype,[])
     |Subst(s) -> ((applysubst sigma(Mkarrow(f1,f2))),s)

let applyfun ((v1,c1),(v2,c2)) =
  let f1 =newvar() in let f2 =newvar() in
  let sigma =
   unifylist((v1,Mkarrow(f1,f2))::(v2,f1)::(c1 @ c2)) in
  match sigma with
      |Fail -> (Notype,[])
      |Subst(s) -> (applysubst sigma f2,s)
```

Astrazione di valori e glb

```
let alfa x = match x with
   | Int _ -> (Intero, [])
   | Bool _ -> (Boolean, [])

let gci ((v1,c1),(v2,c2)) =
    let sigma = unifylist((v1,v2) :: (c1 @ c2)) in
    match sigma with
        |Fail -> (Notype,[])
        |Subst(s) -> (applysubst sigma v1,s)
```

merge e condizionale

```
let valid x = false
let unsatisfiable x = false

let merge (a,b,c) = match a with
   |(Notype,_) -> (Notype,[])
   |(v0,c0) ->
      let sigma = unifylist((v0,Intero)::c0) in
      match sigma with
       |Fail -> (Notype,[])
       |Subst(s) -> match gci(b, c) with
          |(Notype,_) -> (Notype,[])
          |(v1,c1) -> let sigma1 = unifylist(c1@s) in
              match sigma1 with
               |Fail -> (Notype,[])
               |Subst(s1) -> (applysubst sigma1 v1,s1)
```

Calcolo del punto fisso astratto 1

```
let makefunrec (i, Fun(ii, aa), r) =
    let functional ff  =
            sem (Fun(ii, aa)) (bind(r, i, ff)) in
    let rec alfp ff =
      let tnext = functional ff  in
      (match tnext with
         |(Notype, _) -> (Notype,[])
         | _ -> if abstreq(tnext,ff) then ff
                else  alfp tnext  in
    alfp (newvar(), [])
```

Calcolo del punto fisso astratto 2

```
let makefunrec (i, Fun(ii, aa), r) =
    let functional ff  =
            sem (Fun(ii, aa)) (bind(r, i, ff)) in
    let rec alfp ff =
      let tnext = functional ff  in
      (match tnext with
         |(Notype, _) -> (Notype,[])
         | _ -> if abstreq(tnext,ff) then ff
                else  alfp tnext  in
    alfp (newvar(), [])
```

- dato che esistono catene crescenti infinite, il calcolo del punto fisso può divergere
- abbiamo bisogno di un operatore di *widening* che garantisca la terminazione calcolando una approssimazione superiore del minimo punto fisso
  - un tipo meno generale

Widening

```
let makefunrec (i, Fun(ii, aa), r) =
    let functional ff  =
            sem (Fun(ii, aa)) (bind(r, i, ff)) in
    let alfp ff =
      let tnext = functional ff  in
      (match tnext with
         |(Notype, _) -> (Notype,[])
         | _ -> widening(ff, tnext)  in
    alfp (newvar(), [])

let  widening ((f1,c1),(t,c)) =
   let sigma = unifylist( (t,f1) :: (c@c1)) in
   (match sigma with
      |Fail -> (Notype,[])
      |Subst(s) -> (applysubst sigma t,s))
```

- questo widening (unificazione) è usato anche nell'algoritmo di inferenza di tipi di ML