How to transform an analyzer into a verifier

OUTLINE OF THE LECTURE

- a verification technique which combines abstract interpretation and Park’s fixpoint induction
- how to realize a verifier, once you have a "suitable" static analyzer (abstract interpreter)
- experiments using existing analyzers for type domains
  - functional programming à la ML
    - our implementation of a type abstract interpreter in Cousot, POPL 1997
  - logic programming
    - Codish & Lagoon, TCS 2000

THE VERIFICATION METHOD: abstract interpretation

- a semantic evaluation function F_P
  - on a concrete domain $(C, \subseteq)$
  - the least fixpoint $\mathrm{lfp}\ F_P$ is the concrete semantics of program P
- the class of properties we want to verify is formalized as an abstract domain $(A,\leq)$
- $(C, \subseteq)$ and $(A,\leq)$ are related by a Galois connection (α, γ)
- the abstract semantic evaluation function F^{α}_{P} is systematically derived from F_P, α and γ

THE VERIFICATION METHOD: abstract semantics and static analysis

- the abstract semantics lfp F^{α}_{P} is a safe approximation by construction
  - if the property is verified in lfp F^{α}_{P} it is also verified in lfp F_{P}
- static analysis (abstract interpreter) = computation of the abstract semantics lfp F^{α}_{P}
  - effective only if the least fixpoint is reached in finitely many iterations
    - either the abstract domain is Noetherian
    - or we use widening operators

THE VERIFICATION METHOD: partial correctness condition 1

- an element S of the domain (A,≤) is the specification
  - abstraction of the intended concrete semantics
- partial correctness of P wrt S

  $\alpha(\mathrm{lfp}\ F_P) \leq S$

  - not effective since the concrete fixpoint semantics has to computed
- sufficient condition 1
  - for any correct abstract semantic evaluation function F^{α}_P

    $$\mathrm{lfp}\ F^{\alpha}_P \leq S \qquad (1)$$

  - an abstract fixpoint computation is still needed

THE VERIFICATION METHOD: partial correctness condition 2

- an element S of the domain $(A,\leq)$ is the specification
  - abstraction of the intended concrete semantics
- partial correctness of P wrt S

$$\alpha(\mathrm{lfp}\ F_P) \leq S$$

  - not effective since the concrete fixpoint semantics has to computed
- sufficient condition 2 (by fixpoint theorems, abstract version of Park's induction, for any correct abstract semantic evaluation function F^{α}_{P})

$$F^{\alpha}_{P}(S) \leq S \qquad (2)$$

  - no fixpoint computation

PARTIAL CORRECTNESS CONDITIONS

- specification S element of (A,≤)
- sufficient condition 1
  - $$\mathrm{lfp}\ F^{\alpha}_{P} \leq S \qquad (1)$$
    - effective only if (A,≤) is Noetherian or by using widenings
    - stronger than 2 only when widenings are not needed
- sufficient condition 2
  - $$F^{\alpha}_{P}(S) \leq S \qquad (2)$$
    - more efficient (no abstract fixpoint computation)
    - effective even if (A,≤) is non-Noetherian
- ≤ must be decidable
- the specification S must have a finite representation

CAN WE USE AN EXISTING STATIC ANALYZER FOR VERIFICATION?

- condition 1
  - $\text{lfp } F^{\alpha}_{P} \leq S$ (1)
  - straightforward!
  - use the analyzer to compute the abstract semantics
- condition 2
  - $F^{\alpha}_{P}(S) \leq S$ (2)
  - the analyzer must be defined in a denotational style and give access to the function F^{α}_{P}

2 EXAMPLES

- our type inference abstract interpreter for functional programs à la ML
  - the let-polymorphic version
- a type analyzer for logic programs (by Codish & Lagoon) available from a web site

The functional language: syntax

```
type ide = Id of string
type exp =
    | Eint of int
    | Var of ide
    | Sum of exp * exp
    | Diff of exp * exp
    | Ifthenelse of exp * exp * exp
    | Fun of ide * exp
    | Rec of ide * exp
    | Appl of exp * exp
    | Let of ide * exp * exp
    | Letrec of ide * exp * exp
```

The abstract domain of parametric polytypes

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

The meaning of $F^{\alpha}(S^{\alpha}) \leq S^{\alpha}$

- in a language with constructs which create a global environment (typically containing functions),
  - S^{α} is an abstract environment associating to each global name its specification
  - the new expression is evaluated in such an environment
    - assuming that all the global values satisfy their specification
    - using the specification rather than the semantics for the global objects
  - small, modular proofs, which allow us to locate possible bugs
- in our language we have closed expressions only
  - $F^{\alpha}(S^{\alpha})$ is exactly the same as F^{α} for all the syntactic constructs, apart from recursive function definition, where S^{α} (when available, top level) has to be used as first approximation of their abstract value

HOW TO USE THE STATIC ANALYZER FOR TYPE VERIFICATION

`typeinferd: decl→ env→ int→ env`

- was called `sem1` in the lecture on type inference

- compositional verification of a single declaration
- specification S
  - an abstract environment specifying the intended types of
    - global names
    - names defined in the declaration
    - it is finite
- ≤ is the extension to environments of the partial order relation on types

HOW TO USE THE STATIC ANALYZER FOR SUFFICIENT CONDITION 1

```
typeinferd: decl→ env→ int→ env
```

- S type environment
- sufficient condition 1

$$\mathrm{lfp}\ F^{\alpha}_{P} \leq S \qquad (1)$$

```
infercheck (d:decl) (S:env) (n:int) =
     (typeinferd d S n) ≤ S
```

- verification = inference + comparison

HOW TO USE THE STATIC ANALYZER FOR SUFFICIENT CONDITION 2

`typeinferd: decl→ env→ int→ env`

- S type environment
- sufficient condition 2

$$F^{\alpha}_{P}(S) \leq S \qquad (2)$$

  - different for recursive functions only
  - rather than computing (an approximation of) the fixpoint, we evaluate the function expression (once) in the specification
    - we handle recursive functions as standard functions

```
check (d:decl) (S:env) (n:int) =  match d with
   |let id = e -> (typeinferd d S n) ≤  S
   |let rec id = e -> (typeinferd (let id = e) S n) ≤  S
```

HOW TO USE THE STATIC ANALYZER FOR SUFFICIENT CONDITION 2

```
typeinferd: decl→ env→ int→ env
```

- S type environment
- sufficient condition 2

$$F^{\alpha}_{P}(S) \leq S \qquad (2)$$

```
check (d:decl) (S:env) (n:int) =  match d with
   |let id = e -> (typeinferd d S n) ≤ S
   |let rec id = e -> (typeinferd (let id = e) S n) ≤ S
```

- mutual recursion is not shown
- the widening control parameter is used for approximating fixpoints corresponding to recursive functions occurring within e

Why checking $F^\alpha(S^\alpha) \leq S^\alpha$ rather than lfp $F^\alpha \leq S^\alpha$?

- why computing $F^\alpha(S^\alpha)$ rather than lfp F^α ?
- no fixpoint computation
  - more efficient
  - possible even with non-noetherian domains
- modular proofs
  - in which the proof of a component uses the specification rather than the semantics of the other components

TYPE VERIFICATION EXAMPLES 1

- compositionality

```
# check
``let fact = pi id 1''
[pi <- (int -> int) -> int -> int -> int;
    id <- 'a -> 'a;  fact <- int -> int]
1;;
- : bool = true
```

- condition 2 can be better than 1 (widening)

```
# check
``let rec f f1 g n x = if n=0 then g(x) else f(f1)(function x ->
       (function h -> g(h(x)))) (n-1) x f1''
[f <- ('a -> 'a) -> ('a -> 'b) -> int -> 'a -> 'b]
1;;
- : bool = true
# infercheck
``let rec f f1 g n x = if n=0 then g(x) else f(f1)(function x ->
       (function h -> g(h(x)))) (n-1) x f1''
[f <- ('a -> 'a) -> ('a -> 'b) -> int -> 'a -> 'b]
1;;
- : bool = false
```

TYPE VERIFICATION EXAMPLES 2

- let polymorphism

```
# check
``let g = id id''
[ id <- 'a -> 'a; g <- 'b -> 'b ]
1;;
- : bool = true
```

- mutual recursion

```
# check
``let rec ap f x y n = if n=0 then y else ap f x (f x y) (n-1) and
times x n = ap (function z -> function w -> z + w) x 0 n’’
[ap <- ('a -> 'b -> 'b) -> 'a -> 'b -> int -> 'b;
  times <- int -> int -> int]
1;;
- : bool = true
```

TYPE VERIFICATION EXAMPLES 3

- incompleteness

```
# check
``let rec f f1 g n x = if n=0 then g(x) else f(f1)(function x ->
         (function h -> g(h(x)))) (n-1) x f1''
[f <- (int -> int) -> (int -> int) -> int -> int -> int]
1;;
- : bool = false
# infercheck
``let rec f f1 g n x = if n=0 then g(x) else f(f1)(function x ->
         (function h -> g(h(x)))) (n-1) x f1''
[f <- (int -> int) -> (int -> int) -> int -> int -> int]
2;;
-  : bool = true
```

- the specification is not satisfied

```
# check
``let rec f f1 g n x = if n=0 then g(x) else f(f1)(function x ->
         (function h -> g(h(x)))) (n-1) x f1''
[f <- ('a -> 'c) -> ('a -> 'b) -> int -> 'a -> 'b]
1;;
-  : bool = false
```