Combining verification and analysis

CONCLUSIONS ON VERIFICATION

- denotational abstract interpreters have the extra-value of being easily transformed into compositional verifiers
- compositional verification is useful for debugging
  - condition 2 $F^{\alpha}{}_{P}(S) \leq S$

    is exactly the one used in abstract diagnosis to locate possible bugs, when not satisfied
- verification can be combined with analysis (inference), when the program contains property specifications
  - types in ML-like languages

COMBINING VERIFICATION AND ANALYSIS

- the typing rule for recursion in ML

$$\frac{\mathbf{H}\ [\mathbf{f} \leftarrow \tau] \vdash \lambda\mathbf{x.e} \Rightarrow \tau}{\mathbf{H} \vdash \mu\mathbf{f.}\lambda\mathbf{x.e} \Rightarrow \tau}$$

- **H** type environment
- τ monotype with variables
- the expected type σ of the expression can be specified in ML and might be used by the inference algorithm

$$\frac{\mathbf{H}\ [\mathbf{f} \leftarrow \sigma] \vdash \lambda\mathbf{x.e} \Rightarrow \tau \qquad \tau \leq \sigma}{\mathbf{H} \vdash (\mu\mathbf{f.}\lambda\mathbf{x.e} : \sigma) \Rightarrow \sigma}$$

  - the premise of the rule is exactly our condition 2

TYPING RULES AND TYPE CHECKING

- the interesting case is the one of recursion
- the typing rule in the Damas-Milner type system, where **H** is a type environment and τ is a monotype with variables,

$$\frac{\mathbf{H}\ [\mathbf{f} \leftarrow \tau] \mid\!- \lambda \mathbf{x.e} \Rightarrow \tau}{\mathbf{H} \mid\!- \mu \mathbf{f}.\lambda \mathbf{x.e} \Rightarrow \tau}$$

shows that τ is a fixpoint of the functional associated to the recursive definition

- the rule does not give hints on how to guess τ for type inference
- the rule can directly be used for type checking, if τ occurs in the program, as a type specification
- is this rule actually used by the ML's type checking algorithm?

ML's TYPE CHECKER DOES NOT USE THE RECURSION TYPING RULE

$$\frac{H\ [f \leftarrow \tau] \mid\!- \lambda x.e \Rightarrow \tau}{H \mid\!- (\mu f.\lambda x.e : \tau) \Rightarrow \tau}$$

- a counterexample (example 2 with type specification)

```
# let rec (f:('a -> 'a)->('a -> 'b)-> int -> 'a -> 'b)
= function f1 -> function g -> function n -> function x
-> if n=0 then g(x) else f(f1)(function x -> (function
h -> g(h(x)))) (n-1) x f1;;
This expression has type ('a -> 'a) -> 'b but is here
used with type 'b
```

- the specified type is indeed a fixpoint
- suggests that type checking is performed as type inference + comparison (sufficient condition 1, early widening)
- same behaviour with the mutual recursion example

COMBINING VERIFICATION AND ANALYSIS

$$\frac{\mathtt{H}\ [\mathtt{f} \leftarrow \tau]\ |\!-\ \lambda\mathtt{x.e} \Rightarrow \tau}{\mathtt{H}\ |\!-\ (\mu\mathtt{f}.\lambda\mathtt{x.e}:\ \tau) \Rightarrow \tau}$$

- verification of type specifications might help in type inference
  - if the specified type is satisfied, then it is the inferred type
  - more precise types without better fixpoint approximations (no fixpoint computation is involved in type checking)

- we can use a weaker rule for type checking

$$\frac{\mathtt{H}\ [\mathtt{f} \leftarrow \sigma]\ |\!-\ \lambda\mathtt{x.e} \Rightarrow \tau \qquad \tau \leq \sigma}{\mathtt{H}\ |\!-\ (\mu\mathtt{f}.\lambda\mathtt{x.e}:\ \sigma) \Rightarrow \sigma}$$

- the premise of the rule is exactly our condition 2

FROM TYPE SYSTEMS TO TYPE INFERENCE

- type systems are very important to handle a large class of properties
  - functional and object-oriented programming
  - calculi for concurrency and mobility
- the type system directly reflects the property we are interested in
- typing rules are easy to understand
- it is often hard to move from the typing rules to the type inference algorithm
  - systematic techniques are needed
  - abstract interpretation provides some of these techniques