HOW TO TRANSFORM AN ANALYZER INTO A VERIFIER

PART 2

LOGIC PROGRAMMING

LOGIC PROGRAMMING

- condition 2 $F^{\alpha}_{P}(S) \leq S$ is very easy to define if the analyzer is denotational (bottom-up)
  - F^{α}_{P} is the abstract version of the traditional T_P (immediate consequences operator)
    - compositional (for a single clause)
- there exist different denotational semantics, modeling different observables
  - computed answers, call patterns, ...
- most existing analyzers for LP are operational (top-down)
  - not adequate to be transformed into verifiers based on condition 2

A TYPE ANALYZER FOR LOGIC PROGRAMS

- developed by Codish & Lagoon

http://www.cs.bgu.ac.il/~mcodish/Software/aci-types-poly.tgz

- types (abstract terms)
  - set expressions defined by a set constructor +
    - associative, commutative and idempotent
    - terms built from num/0, nil/0, list/1 and variables
- some abstractions

α([X,Y]) = list(X) + list(Y) + nil
α([X|Y]) = list(X) + Y

THE VERIFIER

- developed using the abstract operations of the analyzer

http://www.dimi.uniud.it/~comini/Projects/PolyTypesVerifier/

- the abstract semantics models call patterns in addition to computed answers

- as a consequence, the specification (for a procedure) is a pair of abstract atoms (precondition, postcondition)

- the verification method resulting from the application of condition 2 can be read as

  - the postcondition holds whenever the precondition does and all the procedure calls satisfy their precondition

AN EXAMPLE

- the verifier

verifyIOcall/3: clause * I-spec * O-spec

- a clause for the queens program

```
?- verifyIOcall (
    ( queens(X,Y) :- perm(X,Y), safe(Y)   ),

    [  queens(nil + list(num), T), queens(nil, T),
       perm(nil + list(num), T), perm(nil, T),
       safe(nil + list(num)), safe(nil)   ]  ,

    [  queens(nil, nil),
       queens(nil + list(num), nil + list(num)),
       perm(nil, nil),
       perm(nil + list(num), nil + list(num)),
       safe(nil + list(num)), safe(nil)   ]   ).
No.1 : yes
```

- if we change the order of atoms in the clause body (same specifications)

```
?- verifyIOcall (
    ( queens(X,Y) :- safe(Y),  perm(X,Y)),
    [  …   ]  ,
    [  …   ]   ).
Clause may be wrong because call safe(U) (atom
number 1 of body) is not in the call-specification.
```