Regular Types, Modes, and Model Checking

John Gallagher
University of Roskilde, Denmark

Collaborators
Kim Henriksen, Roskilde
Framework 5 IST ASAP project partners
Univ. Politécnica de Madrid
Southampton Univ.
Bristol Univ.

Computer Science, building 42.1
Roskilde University
Universitetsvej 1
P.O. Box 260
DK-4000 Roskilde
Denmark
Phone: +45 4674 2000
Fax: +45 4674 3072
www.dat.ruc.dk

Computer Science
Roskilde University

Background - mode analysis

- Instantiation modes
  - perhaps the earliest static analysis of logic programs (Mellish, Bruynooghe,... mid 1980s)
  - abstract substitutions by mode substitutions (e.g. ground, partly instantiated, free)
  - abstract unification algorithm to propagate modes
  - mostly interested in modes of call patterns, hence goal-directed interpretation.
  - derivation of modes like p(+,+,-,?) allows compiler optimisations

Booleans and constraints for modes

- The system TOUPIE (Corsini et al.) took a different approach
- Mode analysis seen as a boolean constraint solving problem.
- Given equality $X=f(Y_1,\ldots,Y_n)$ in the program one could associate a constraint (after unifying the terms)
  - X is ground iff Y_1 is ground and .... and Y_n is ground, represented simply as $X \leftrightarrow Y_1 \wedge \ldots \wedge Y_n$
  - Note that this is a success mode, not a call mode.

Boolean constraints (continued)

- Codish and Demoen exploited a similar idea, but using explicit values true and false, and encoding the boolean groundness dependencies using atomic formulas
  - e.g. X = [Y|Z] is replaced by a formula iff(X,Y,Z) with the definition

```
iff(true,true,true).
iff(false,true,false).
iff(false,false,true).
iff(false,false,false).
```

essentially the truth table for $X \leftrightarrow Y \wedge Z$

Abstract programs

- After replacing equalities by iff atoms, Codish and Demoen obtained an abstract program.
- Computing the least model gave an abstract success set for the program predicates.
  - E.g. for append, they obtained the model
    {append(true,true,true), append(true,false,false), append(false,true,false),append(false,false,false)}
    This a a representation of the boolean dependency append(X,Y,Z) = $X \wedge Y \leftrightarrow Z$

More boolean dependencies

- Codish and Demoen then showed that boolean dependencies could encode other properties than groundness.
- E.g. considering the equality X=[Y|Z] one can see that (after successfully unifying)
  - "X is a list iff Z is a list", represented as iff(X,Z) where iff(true, true). iff(false,false).
  - Similarly X=[] is replaced by iff(X) where iff(true).
- Compute the least model for the abstract program for append
  - {append(true,true,true), append(true, false, false)}
  - which represents append(X,Y,Z) = $X \wedge Y \leftrightarrow Z$
  - "Z is a list iff both X and Y are lists"

Pre-interpretations

- Boulanger, Bruynooghe and Denecker
  - Consider the language of a program P. Standard semantics is based on Herbrand pre-interpretation, which has the Herbrand universe U_P as domain as interpretation.
  - Semantics is the least model M_P.
  - U_P is the most precise domain possible
  - One can define an abstraction by defining a pre-interpretation J over some more abstract domain D_J.
  - Then compute a model M_J based on J.
  - Any atom true in M_P is also true in any other model.
  - Hence M_J represents a safe approximation of M_P.

Capturing modes in a pre-interpretation

- It seems at first sight difficult to capture modes in a "ground semantics".
- One can regard the language as containing extra constants {v1,v2,...} which represent variables
- The minimal model contains all atomic logical consequences (the Clark semantics)
  - occurrences of v1,v2,... in the minimal Herbrand model are isomorphic to the occurrence of variables in true atoms.

The groundness pre-interpretation

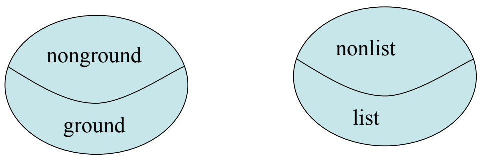
- Consider a domain of interpretation {g,ng}
- Given function symbols {[]/0, [.|.]/2, a/0, v/0}
- We assume that v is non-ground
- Pre-interpretation is
  - [] → g
  - a → g
  - v → ng
  - [g|g] → g
  - [g|ng] → ng
  - [ng|g] → ng
  - [ng|ng] → ng
- The least model of append over this interpretation is
  {append(g,g,g), append(g,ng,ng),append(ng,g,ng), append(ng,ng,ng)}
  Just the same as the boolean dependencies.

The list pre-interpretation

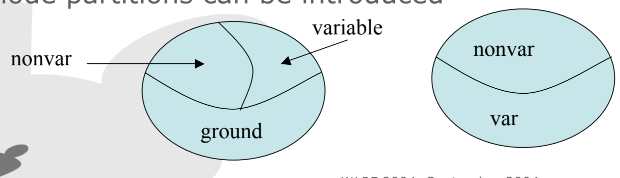
- Consider a domain of interpretation {list, nonlist}
- Given function symbols {[]/0, [.|.]/2, a/0}
- Pre-interpretation is
  - [] → list
  - a → nonlist
  - [list|list] → list
  - [nonlist|list] → list
  - [list|nonlist] → nonlist
  - [nonlist|nonlist] → nonlist
- The least model of append over this interpretation is
  {append(list,list,list), append(list,nonlist,nonlist)}
  Just the same as the boolean dependencies.

New mode analyses

- The elements of the pre-interpretation can be thought of as partitioning the set of all terms.



- Other mode partitions can be introduced



The fgi pre-interpretation

- Use the three domain elements {f,g,i}
  - free, ground, and partly instantiated
- Pre-interpretation over {[]/0, [.|.]/2, a/0}

[] → g	[f|f] → i
a → g	[i|f] → i
[g|g] → g	[g|i] → i
[f|g] → i	[f|i] → i
[i|g] → i	[i|i] → i
[g|f] → i	

Model of append
append(g,g,g)
append(g,v,v)
append(g,i,i)
append(g,v,i)
append(i,g,i)
append(i,v,i)
append(i,i,i)

Note that we cannot summarise the model as a boolean formula.

Pre-interpretations plus abstract compilation

- Pre-interpretations were “compiled in” to a program
- Gallagher-Boulanger-Saglam (ILPS-1995)
- Various simple mode and type analysis shown
- Infinite pre-interpretations (e.g. size-norms) could also be handled in this approach.

Type Analysis (set based analysis)

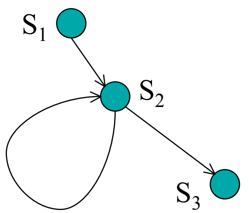
Aim of <u>set-based analysis</u> - to find a conservative approximation of the set of values that can appear at a given program point (work goes back to [Reynolds, 1968])

```
q([],X,X).
q([c(X1)|Y],Acc,X) ←
        integer(X1), q(Y,c(X1,Acc),X).
q([d(X1)|Y],Acc,X) ←
        integer(X1), q(Y,d(X1,Acc),X).
p(X,Y) ← q(X,0,Y).
```

S_Y ::= 0 | c(Int, S_Y) | d(Int, S_Y)

(S_Y is an infinite regular set of terms)

Set-based Analysis for Specialization




```
s1(X) ← action1(X,Y), s2(Y).
s2(X) ← action2(X,Y), s2(Y).
s2(X) ← action3(X,Y), s3(Y).
```

Problem - to get an accurate specialization of s_3.

Example: When specializing interpreter for procedure calls, approximate the stack, otherwise continuation code is unknown.

```
exec([call(p(N))|Cont],Stack) ←
   code(p(N),Pcode),
   push(Cont,Stack, Stack1),
   exec(Pcode,Stack1).
. . . .
exec([return],Stack) ←
   pop(Stack, ContCode,Stack1),
   exec(ContCode,Stack1).
```


Regular Approximation of Data Structures

Stack	→ cons(Pcont,S_1) | cons(Rcont,S_2)
S_1	→ cons(Qcont,Stack)
S_2	→ emptyStack

```
. . .
call r;
. . .

proc r {
  . . .
  call p;
  . . . }

proc p {
  if e {return}
  else call q;
  . . . }

proc q {
  . . .
  call p; }
```

Stack = (Pcont Qcont)*Rcont

In general, non-deterministic tree grammars are required to represent such structures.

Set-Based Analysis

- There are several approaches to set-based analysis
  - Derive *set constraints* from the program text and solve the constraints [Heintze & Jaffar]
  - *Abstract interpretation* of the program over a domain of regular types [Jones, Dart & Zobel, Janssens & Bruynooghe, Gallagher & de Waal, van Hentenryck et al., …]
  - Approximate the (logic) program by a *monadic "type" program*, and then transform that program to a normal form [Frühwirth et al.].

Non-Deterministic Finite Tree Automata (Non-deterministic tree grammars)

Tree automata provide a means of specifying infinite sets of trees (terms) over some signature $\sum$.

A tree automaton over $\sum$ is a tuple $<Q, q^*, \Delta>$ where

Q is a finite set of states

$q^* \in Q$ is an accepting state

Δ is a finite set of transitions of the form

$f(q_1,\ldots,q_n) \rightarrow q_0$,

where $q_0, q_1,\ldots,q_n \in Q$, and f is an n-ary function in $\sum$.

Example: $q^* = S_Y$, transitions $\{0 \rightarrow S_Y, c(Int, S_Y) \rightarrow S_Y, d(Int, S_Y) \rightarrow S_Y\}$

Non-Deterministic Finite Tree Automata (NFTAs)

A tree automaton is (top-down) *non-deterministic* if contains two transitions with

the same right-hand state q_0, and
the same function f on the left-hand-side.

Example: {[] → As, [A|As] → As, [] → Bs, [B|Bs] → Bs,
[] → S, [A|As] → S, [B|Bs] → S, a → A, b → B}

Accepting state S represents the union of As (the set of lists of element a), with Bs (the set of lists of element b).
{[], [a], [a,a],…, [b],[b,b],[b,b,b],…}
The non-deterministic transitions are highlighted.

Limited Precision of Deterministic Grammars

append([], Ys,Ys).
append([X|Xs], Ys, [X|Zs]) ← append(Xs,Ys,Zs).

?- append(A,B,C).

?

[] → A
[a | A] → A

[a,a,….a]

[] → B
[b | B] → B

[b,b,….b]

with a *deterministic* automaton, the best we can do is
[] → C
[D | C] → C
a → D
b → D
This is the set of lists of a and b (mixed).
[a,a,b,a,b,b,….a]

Increased Precision of Non-Determinism

With NFTAs, we can describe a more precise result.

[] → C
[a | C] → C
[b | B] → C
[] → B
[b | B] → B

[a,a,a,….,b,b,b]   sequence of ‘a’ *followed by* sequence of ‘b’

The extra precision can be used for more accurate debugging, specialisation, verification etc.

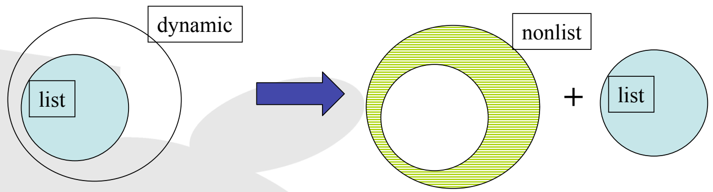
Analysis For Non-Deterministic Descriptions

- Set-constraint approaches yield non-deterministic descriptions
- Previous abstract interpretations used only deterministic descriptions

- Our aim: to achieve *the precision of set-constraints* within the *flexible framework of abstract interpretation* (first suggested by Cousot & Cousot 1995).

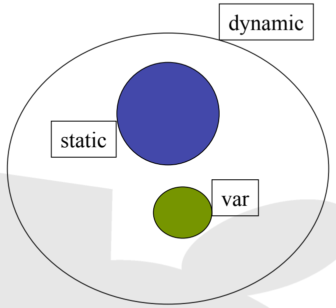
Determinisation of FTAs

- Any FTA can be determinised.
- There is an equivalent FTA (defining the same sets of terms) that is bottom-up deterministic
- In a deterministic FTA, each term is in at most one type (state). Types are disjoint.



Modes as Types

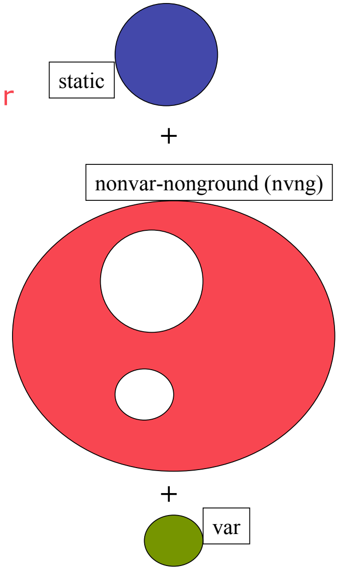
- Modes are also types
- Add an extra constant $VAR to the language (which is defined to be non ground)
- Define types *var*, *static (or ground)* and *dynamic*.



<u>Transitions</u>
a → static
b → static
f(static,...,static) → static
[static|static] → static
. . .
a → dynamic
b → dynamic
f(dynamic,..., dynamic) → dynamic
[dynamic | dynamic] → dynamic
$VAR → dynamic
. . .
$VAR → var

Determinised modes

- Modes static, dynamic and var
  - [] → static
  - a → static
  - b → static
  - [static|static] → static
  - f(static,...,static) → static
  - . . .
  - [var|*] → nvng
  - [nvng|*] → nvng
  - f(*,...,var,...,*) → nvng
  - f(*,...,nvng,...,*) → nvng
  - . . .
  - $VAR → var



Determinisation of list/dynamic

[] → list
[list|list] → list
[nonlist|list] → list
[nonlist|nonlist] → nonlist
[list|nonlist] → nonlist
a → nonlist
b → nonlist
f(*,*,...,*) → nonlist
. . .

Writing this as “types” for list and nonlist
list = []; [nonlist|list]; [list|list]
nonlist = a; b; [nonlist|nonlist]; [list|nonlist]; f(*,...,*);...

Computer Science Roskilde University
WLPE 2004, September 2004

Abstract compilation of a pre-interpretation

1. Put each clause in normal form
   - every argument of predicates (apart from =/2) is a variable
   - every equality atom is of the form $f(X_1,\ldots,X_n)=X_0$

   Example
   append(U,X,X) :- []=U.
   append(U,Y,V) :- append(Xs,Y,Zs), [X|Xs]=U, [X|Zs]=V.
   reverse(U,V) :- []=U, []=V.
   reverse(U,V) :- reverse(Xs,W),append(W,Z,V), [X|Xs]=U, $[X|X_1]=Z$, $[]=X_1$.

2. Then replace = by →. The predicate → is defined by a pre-interpretation (determinised FTA).

Computer Science Roskilde University
WLPE 2004, September 2004

Least model wrt to a pre-interpretation

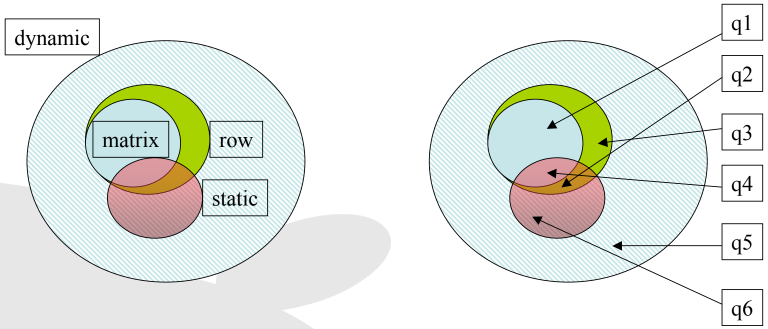
- The least model of the transformed program P is $lfp(T_P)$
- The arguments of the predicates (apart from →) are domain elements (types).
- E.g. using the domain {list, nonlist} and the determinised transitions, the least model is

reverse(list, list)
append(list, nonlist, nonlist), append(list, list, list)

Computer Science Roskilde University
WLPE 2004, September 2004

Mixing modes and types in BTA

- Binding time analysis in off-line partial evaluation
- Static, dynamic and program-specific types



Summary - regular-type-based analysis

1. Define some regular types
2. Determinise the corresponding FTA, obtaining a pre-interpretation
3. Compute the minimal model wrt to the pre-interpretation

Model Checking

- Checking whether given program-specific properties hold (at some program point)
- Reasoning over all reachable states

- For finite state systems, complete exhaustive coverage is possible

- For infinite state systems we abstract the state space

Even-odd

```
odd_even:- even(X), even(s(X)).

even(zero).
even(s(X)):- odd(X).
odd(s(X)):- even(X).
```

Can even_odd succeed? We have seen above two approaches:

1. Compute a regular approximation
2. Compute a pre-interpretation over the abstract values {even, odd} defined by
   zero → even. s(even) → odd. s(odd) → even.

lists of as and bs

```
[] -> alist.
[a|alist] -> alist.

[] -> blist.
[b|blist] -> blist.

[a|blist] -> ablist.
[a|ablist] -> ablist.

[b|alist] -> balist.
[b|balist] -> balist.

a -> a.
b -> b.
```

show that reversing an ablist yields a balist.

Computer Science Roskilde University

Infinite State Model Checking

Prolog program representing operations on a token ring (with any number of processes) (example from Podelski & Charatonik).

```
gen([0,1]).
gen([0 | X]) ← gen(X).
trans(X,Y) ← trans1(X,Y).
trans([1 |X],[0|Y]) ← trans2(X,Y).
trans1([0,1|T],[1,0 |T]).
trans1([H|T],[H|T1]) ← trans1(T,T1).
trans2([0],[1]).
trans2([H|T],[H|T1]) ← trans2(T,T1).
reachable(X) ← gen(X).
reachable(X) ← reachable(Y), trans(Y,X).
```

```
0 -> zero.
1 -> one.
[] -> zerolist.
[zero|zerolist] -> zerolist.
[one|zerolist] -> goodlist.
[zero|goodlist] -> goodlist.
```

```
% q3 = [dynamic]
% q1 = [dynamic,goodlist]
% q4 = [dynamic,one]
% q5 = [dynamic,zero]
% q2 = [dynamic,zerolist]
[reachable(q1)].
[trans(q1,q1),trans(q3,q3)].
[trans1(q1,q1),trans1(q3,q3)].
[trans2(q1,q3),trans2(q2,q1),
  trans2(q3,q3)].
```

Computer Science Roskilde University

Cryptographic Protocol Example (Blanchet)

```
attacker(pencrypt(M,PK)) ← attacker(M),attacker(PK).
attacker(pk(SK)) ← attacker(SK).
attacker(M) ← attacker(pencrypt(M,pk(SK))), attacker(SK).
attacker(sign(M,SK)) ← attacker(M), attacker(SK).
attacker(M) ← attacker(sign(M,SK)).
attacker(sencrypt(M,K)) ← attacker(M), attacker(K).
attacker(M) ← attacker(sencrypt(M,K)), attacker(K).
attacker(pk(skA)).
attacker(pk(skB)).
attacker(a).
attacker(pencrpyt(sign(k(pk(X)),skA),pk(X))) ← attacker(pk(X)).
attacker(sencrpyt(s,K1)) ← attacker(pencrpyt(sign(K1,skA),pk(skB))).
unsafe ← attacker(s).   (unsafe state:  if attacker gets the secret)
```

```
Abstraction of Denning-Sacco Protocol (by B. Blanchet)
pencrypt(M,PK): encrypt message M with private key PK.
pk(SK): public key built from secret key SK.
sign(M,SK): message M signed with secret key SK.
sencrypt(M,K): encrypt message M with shared key K.
```

Computer Science Roskilde University

FTAs and Model checking

- FTAs (regular types) provide an expressive notation for specifying program properties
  - how expressive? links to CTL?
- FTAs can also capture properties of tuples (tree-tuple languages) by suitable codings
- A general approach has been outlined for converting a set of given regular types into a program analysis
- Now the "only" problem is complexity!

Computer Science Roskilde University