Program Analysis and Transformation based on Tree Automata

John Gallagher
University of Roskilde, Denmark

Supported by Framework 5 IST Project ASAP

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

Motivating examples (1)

```
oddEven ← even(X), even(s(X)).

even(0).
even(s(X)) ← odd(X).
odd(s(X)) ← even(X).

Can the query oddEven succeed?
```

```
main(X) ←
   zeroList(X), .....,
   member(1,X).

zeroList([]).
zeroList([0|X]) ← zeroList(X).

member(X,[X|_]).
member(X,[_|Y]) ← member(X,Y).

Can the query main(X) succeed?
```


Motivating examples (2)

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).
```

What are the possible answers for reachable(X)? Can X be a list containing more than one '1'?

```
gen([0,1]).
gen([0,0,1) .
gen([0,0,0,...,1]).
....
```

Intended reachable states
reachable([0,0,...,1,...0,0])
(lists with exactly one 1)

Motivating Examples (3)

```
/* transpose a matrix */

transpose(Xs,[]) :-
        nullrows(Xs).
transpose(Xs,[Y|Ys]) :-
        makerow(Xs,Y,Zs),
        transpose(Zs,Ys).

makerow([],[],[]).
makerow([[X|Xs]|Ys],[X|Xs1],[Xs|Zs]):-
        makerow(Ys,Xs1,Zs).


nullrows([]).
nullrows([[]|Ns]) :-
        nullrows(Ns).
```

```
row --> []; [any | row]

matrix --> []; [row | matrix]
```

Show "type correctness" of transpose(X,Y) . I.e. X and Y are both of type "matrix" in all possible solutions.

Show "mode correctness" of transpose(X,Y) . I.e. X is a ground term iff Y is a ground term.

Motivating Examples (4)

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).
```

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

Show that all solutions of reachable(X) are such that X is a goodlist.

Motivating Examples (5)

```
/* transpose a matrix */

transpose(Xs,[]) :-
        nullrows(Xs).
transpose(Xs,[Y|Ys]) :-
        makerow(Xs,Y,Zs),
        transpose(Zs,Ys).

makerow([],[],[]).
makerow([[X|Xs]|Ys],[X|Xs1],[Xs|Zs]):-
        makerow(Ys,Xs1,Zs).


nullrows([]).
nullrows([[]|Ns]) :-
        nullrows(Ns).
```

```
row --> []; [any | row]

matrix --> []; [row | matrix]
```

Suppose we are <u>partially evaluating</u> transpose(X,Y) w.r.t a partially known matrix, where X is a list of unknown values, e.g. X = [U1,U2,U3]. I.e. specialise for 3 X m matrices.

Show that every call to transpose during partial evaluation has its first argument instantiated to a list.

Approximating sets of terms

- Let Σ be a signature - a set of function symbols, each having a rank (arity)
- Term(Σ) is the set of all terms (trees) constructible from Σ
  - i.e. terms of form $f(t_1,\ldots,t_n)$ where $f \in \Sigma$, f has arity n and $t_1 \in$ Term(Σ),…,$t_n \in$ Term(Σ)
  - when arity is 0, we write f() as f.
- Term^n(Σ) denotes the set of n-ary relations over Term(Σ).

Regular/Recognizable Tree Languages

- Suppose Σ = {[], [.|.], 0, s(.)}
- We can specify the set of all lists, i.e.
  {[],[0],[s(0)],[s(s([])), 0], [[]], [[0],[0,0]],...}

```
[] --> list
[any|list] --> list

0 --> any
[] --> any
[any|any] --> any
s(any) --> any
```

NFTA - Nondeterministic finite tree automata

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

A (nondeterministic) finite tree automaton (N)FTA is a tuple <Q, Q_f, Σ, Δ> where
  Q is a finite set of states
  $Q_f \subseteq Q$ are the accepting states
  Δ is a finite set of transitions (rules) of the form
    f(q1,…,qn) → q0,
    where q0, q1,…,qn ∈ Q, and f is an n-ary function in Σ.

An FTA A defines a set of terms L(A) (we will see how shortly)

Example: <{list, any}, {list}, {[], [.|.], 0, s(.)}, Δ>
where Δ = {[] → list,[any|list] → list, 0 → any,[] → any,
  [any|any] → any, s(any) → any}

Cartesian Approximation

- Our aim is to approximate the relations computed by logic programs.
- Let R be some relation over Term(Σ)
- The Cartesian approximation of a relation R is the product of the sets of values in each position of the relation.
  - E.g. let R = reverse = {<[],[]>, <[a],[a]>, <[a,b],[b,a]>, <[a,a,b],[b,a,a]>,...},
  - or written as {reverse([],[]), reverse([a],[a]), reverse([a,b],[b,a]), reverse([a,a,b],[b,a,a]), ...}
- Cartesian approximation is R_1X R_2 where R_1 = {[], [a], [a,b], [a,a,b],....} and R_2 = {[], [a], [b,a], [b,a,a],....}

Approximation Using FTAs

- The set of values in each argument will be approximated using an FTA.
- So we could approximate reverse as reverse = {<x,y> | x ∈ L(A), y ∈ L(A)} where A is the FTA <{list,a,b},{list},Σ,Δ>
  - Σ = {[],[.|.],a,b}, Δ = {[]→list, [a|list]→list, [b|list]→ list, a→a, b→b}
- So reverse has lists of a and b as arguments.
  - we write reverse(list, list) as the approximation.
  - in general, we write a Cartesian approximation of relation R using FTAs as R(q1,...,qn) where q1,...,qn are the states in an FTA.

Two Approaches to Analysis using FTAs

1. Given a program and an FTA, compute an approximation of the program in terms of the states in the given FTA.
   - e.g. given the matrix transpose program and the FTA defining matrices, derive the relation transpose(matrix,matrix) as an approximation.
2. Given a program, derive an FTA that is a safe approximation of the relations defined by the programs
   - e.g. given the reverse program, derive the list-FTA and the relation approximation reverse(list,list).

FTA Properties and Operations

- FTAs form a reasonably expressive language for describing sets of terms.
- Languages defined by FTAs are closed under operations (intersection, union, complement).
- Emptiness of an FTA and membership of a term in L(A) are decidable.

- We will see later that expressiveness can be increased more, while retaining desirable computational properties.

Running an FTA

- Top-down

1. Initialise current term = an accepting state
2. Pick a state q at a leaf in the current term, and find a rule f(q1,…,qn) → q
3. Replace q by f(q1,…,qn)
4. Terminate (successfully) when a term in Term(Σ) is generated

- Bottom-up

1. Initialise current term = a term in Term(Σ)
2. Pick a subterm f(q1,…,qn) from the current term, and find a rule f(q1,…,qn) → q
3. Replace f(q1,…,qn) by q
4. Terminate (successfully) when the current term is an accepting state.

Running the list-FTA

- Top-down
  - list
    replace list by [any|list]
  - [any|list]
  - [s(any)|list]
  - [s(s(any))|list]
    replace any by 0
  - [s(s(0))|list]
  - [s(s(0)), any|list]
  - [s(s(0)), 0|list]
    replace list by []
  - [s(s(0)), 0]

- Bottom-up
  - [s(s(0)), 0]
    replace [] by list
  - [s(s(0)), 0|list]
  - [s(s(0)), any|list]
  - [s(s(0))|list]
    replace 0 by any
  - [s(s(any))|list]
  - [s(any)|list]
  - [any|list]
    replace [any|list] by list
  - list

Language accepted by an FTA

- Top-down and bottom-up are equivalent
- Given an FTA <Q,Q_f, Σ,Δ>
  - there exists a top-down run (derivation) from accepting state $q \in Q_f$ to $t \in$ Term(Σ) if and only if there exists a bottom-up run (derivation) from t to q.
- In either case we say that t is accepted by (state q of) the FTA.

- The set of all terms accepted by some final state of an FTA A is called the language of A, L(A).

Regular tree languages

- If a set of terms can be represented as L(A) from some FTA A, we say that the set of terms is recognizable.
- Such a set of terms is also known as a regular tree language
  - the set Δ can be seen as a regular tree grammar.

Deterministic FTAs

- Unlike string automata, determinism comes in two flavours.
  - An FTA is bottom-up deterministic (DFTA) if there are no two rules in Δ having the same left-hand-side.
    - f(q1,...,qn) $\rightarrow$ q and f(q1,...,qn) $\rightarrow$ q', q $\neq$ q' disallowed
  - An FTA is top-down deterministic (DTTA) if there are no two rules in Δ having both the same right-hand-side and the same function symbol on the left.
    - f(q1,...,qn) $\rightarrow$ q and f(s1,...,sn) $\rightarrow$ q, qi $\neq$ si disallowed

Equivalence of FTAs and DFTAs

- For every FTA, there is an equivalent DFTA (bottom-up deterministic FTA).
- However, this does not hold for top-down deterministic FTAs.
  - there are some FTAs that have no equivalent DTTA.
  - E.g. Σ = {[],[.|.],a,b}, Δ = {[]$\rightarrow$ablist, [a|ablist] $\rightarrow$ablist, [b|ablist]$\rightarrow$blist, []$\rightarrow$blist, [b|blist]}
  - (lists of a's followed by b's, [a,a,a,....,b,b,b])

Disjoint Accepting States in DFTAs

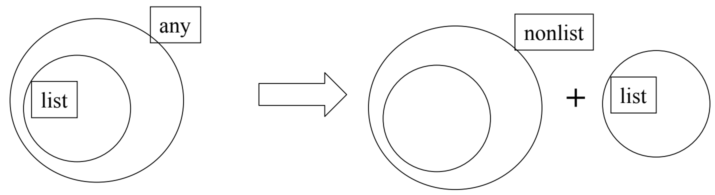
- Given a DFTA and a term t, we can see that a bottom-up run starting from t is deterministic.
- Hence each term can be accepted by at most one state of a DFTA.
- Thus the sets of terms accepted by the states of a DFTA are disjoint.

Determinizing FTAs

- An algorithm exists for converting an arbitrary FTA to a DFTA.
- Consider transitions for list and any
  - [] → list
  - [any|list] → list
  - [] → any
  - [any|any] → any
  - 0 → any
  - s(any) → any
- This is not b-u deterministic ([] occurs twice in lhs of a transition)

Determinization of FTAs

- Any FTA can be determinized.
- There is an equivalent FTA that is bottom-up deterministic
- In a deterministic FTA, each term is in at most one type (state). Types are disjoint.



Determinization of list/any

[] → list'
[list'|list'] → list'
[nonlist|list'] → list'
[nonlist|nonlist] → nonlist
[list|nonlist] → nonlist
0 → nonlist
s(list) → nonlist
s(nonlist) → nonlist

list' = [list ∩ any]
nonlist = [any]

An expression [q1, q2, ...,qn] denotes a state in the DFTA that accepts terms accepted by all of q1,...,qn and *accepted by no other state*.

Advantages of DFTAs for approximation

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

The best approximation of the append relationusing the FTA defining list and any.

append(list, any, any).
append(list, any, list).
append(list, list, any).
append(list, list, list).

⇨ append(list, any, any).

The first argument is definitely a list, but *no dependencies* between the second and third arguments can be detected. This is because list and any are not disjoint.

Approximation using DFTA

We list the minimum set of true “abstract facts” for append over the determinized types list' and nonlist.

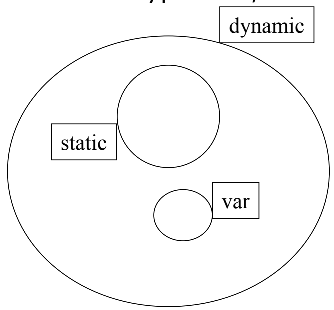
append(list', list', list').
append(list', nonlist, nonlist).

The first argument has to be a list, and the *dependency between the second and third arguments can be observed*.

(Similar analysis performed using Boolean abstract domain (Codish-Demoen)).

Modes defined by FTAs

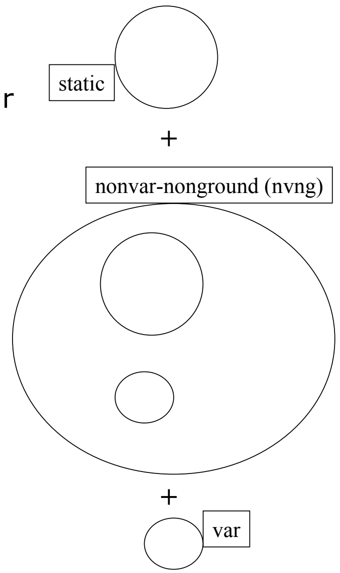
- Instantiation modes (like ground, nonvar, var) can also be defined by FTAs
- Add an extra constant $VAR to the language (which is defined to be non-ground)
- Define types *var*, *static (or ground)* and *dynamic*.



<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

Determinized 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

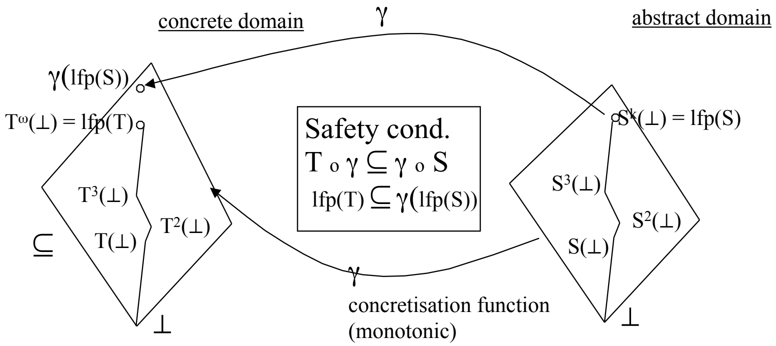


From DFTAs to abstract interpretation

- A determinized automaton can be seen as a pre-interpretation of a given set of constants and functions.
- E.g. the set D = {static, var, nvng} is the domain of a pre-interpretation
- The determinized mode transitions define functions
  - for each n-ary functor f, the transitions define a function $D^n \to D$

Abstract Interpretation of Logic Programs

- Aim is to approximate the semantics of a logic program.
- The minimal Herbrand model is the concrete semantics.
- It is the least fixed point of the "immediate consequence operator" T_P



Computing the model of a program

oddEven ← even(X), even(s(X)).

even(0).
even(s(X)) ← odd(X).
odd(s(X)) ← even(X).

Initial approximation = ∅

$T(\emptyset)$ = {even(0)}
$T^2(\emptyset)$ = {even(0), odd(s(0))}
$T^3(\emptyset)$ = {even(0), odd(s(0)), even(s(s(0)))}
.....

Minimal model of the program is the limit of this sequence, which is the least fixedpoint of T.

oddEven will not be in the least fixpoint, but this requires an inductive proof.

Abstraction using even-odd types

- Consider the FTA <{e,o}, {e,o}, {0,s(.)}, Δ}, where
  - Δ = {0 → e, s(e) → o, s(o) → e}
- This is already a DFTA so does not need determinizing.
- We will compute the least model with this pre-interpretation.

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 some pre-interpretation (determinized FTA).

Abstraction of the even-odd program

oddEven ← even(X), even(U), s(X)→U. even(U) ← 0→U even(U) ← s(X)→U, odd(X). odd(U) ← s(X)→U, even(X). 0→e. s(e)→o. s(o)→e.	Computing the model Initial approximation = ∅ $T(\emptyset)$ = {0→e,s(e)→o,s(o)→e} $T^2(\emptyset)$ = {0→e,s(e)→o,s(o)→e, even(e)} $T^3(\emptyset)$ = {0→e,s(e)→o,s(o)→e, even(e),, odd(o)} $T^3(\emptyset) = T^4(\emptyset)$ We can see that oddEven has no solution in the abstract model. Hence it has no solution in the concrete model either.

Abstract program - another 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.

[] → list.
[list|list] → list.
[nonlist|list] → list.
[nonlist|nonlist] → nonlist.
[list|nonlist] → nonlist.

This program has a finite least model

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

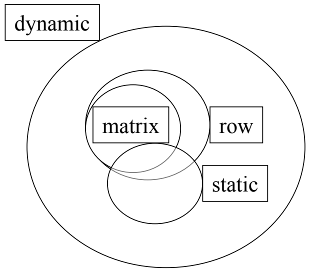
reverse(list, list),
append(list, nonlist, nonlist),
append(list, list, list)

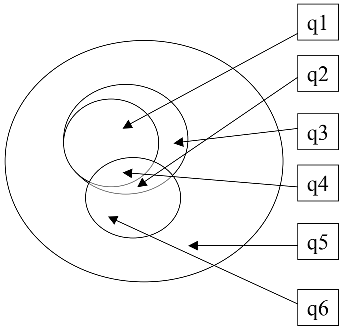
Steps in building a regular-type-based analysis

- Define some regular types
- Determinise the corresponding FTA, obtaining a pre-interpretation
- Compute the minimal model wrt to the pre-interpretation
  - we use abstract compilation and then compute minimal Herbrand model of abstract program

Mixing modes and types in BTA

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





Determinizing modes+lists example

% q1 = [dynamic ∩ matrix ∩ row]
% q2 = [dynamic ∩ row ∩ static]
% q3 = [dynamic ∩ row]
% q4 = [dynamic ∩ matrix ∩ row ∩ static]
% q5 = [dynamic]
% q6 = [dynamic ∩ static]
$VAR -> q5.
$CONST -> q6.
[A|q5] -> q5.
[A|q3] -> q3.
[q2|q4] -> q4.
[q4|q4] -> q4.
[q2|q6] -> q6.
[q4|q6] -> q6.
[q2|q2] -> q2.
[q4|q2] -> q2.
[q2|q1] -> q1.
[q4|q1] -> q1.
[q5|q4] -> q3.
[q5|q6] -> q5.
[q5|q2] -> q3.
[q5|q1] -> q3.
[q6|q4] -> q2.
[q6|q6] -> q6.
[q6|q2] -> q2.
[q6|q1] -> q3.
[q1|q4] -> q1.
[q3|q4] -> q1.
[q1|q6] -> q5.
[q3|q6] -> q5.
[q1|q2] -> q3.
[q3|q2] -> q3.
[q1|q1] -> q1.
[q3|q1] -> q1.
[] -> q4.

Analyzing Programs using disjoint types

- E.g. for naive reverse, with above pre-interpretation:

{app(q3,q6,q5),app(q3,q5,q5),app(q3,q4,q3),
app(q3,q3,q3),app(q3,q2,q3),app(q3,q1,q3),
app(q2,q6,q6),app(q2,q5,q5),app(q2,q4,q2),
app(q2,q3,q3),app(q2,q2,q2),app(q2,q1,q3),
app(q1,q6,q5),app(q1,q5,q5),app(q1,q4,q1),
app(q1,q3,q3),app(q1,q2,q3),app(q1,q1,q1),
app(q4,A,A)}

{rev(q4,q4),rev(q3,q3),rev(q2,q2),rev(q1,q1)}
Compact representations are essential!

Precision

- The method computes the least model with the given pre-interpretation (DFTA).
- Accurate query-dependent information can be obtained by querying the model.
  - module at a time analysis without loss of precision
  - “condensing” property
  - call patterns can be computed by a separate fixpoint iteration

Steps in building an FTA-based analysis

- Define an FTA capturing some properties of interest
- Determinize the FTA, obtaining a pre-interpretation (DFTA)
- Compute the minimal model wrt to the pre-interpretation
  - use abstract compilation and then compute minimal model of abstract program

Determinizing modes+lists example

% q1 = [dynamic ∩ matrix ∩ row]
% q2 = [dynamic ∩ row ∩ static]
% q3 = [dynamic ∩ row]
% q4 = [dynamic ∩ matrix ∩ row ∩ static]
% q5 = [dynamic]
% q6 = [dynamic ∩ static]
$VAR -> q5.
$CONST -> q6.
[A|q5] -> q5.
[A|q3] -> q3.
[q2|q4] -> q4.
[q4|q4] -> q4.
[q2|q6] -> q6.
[q4|q6] -> q6.
[q2|q2] -> q2.
[q4|q2] -> q2.
[q2|q1] -> q1.
[q4|q1] -> q1.
[q5|q4] -> q3.
[q5|q6] -> q5.
[q5|q2] -> q3.
[q5|q1] -> q3.
[q6|q4] -> q2.
[q6|q6] -> q6.
[q6|q2] -> q2.
[q6|q1] -> q3.
[q1|q4] -> q1.
[q3|q4] -> q1.
[q1|q6] -> q5.
[q3|q6] -> q5.
[q1|q2] -> q3.
[q3|q2] -> q3.
[q1|q1] -> q1.
[q3|q1] -> q1.
[] -> q4.

Analyzing reverse using disjoint modes + types

- E.g. for naive reverse, with above pre-interpretation:

  {app(q3,q6,q5),app(q3,q5,q5),app(q3,q4,q3),
  app(q3,q3,q3),app(q3,q2,q3),app(q3,q1,q3),
  app(q2,q6,q6),app(q2,q5,q5),app(q2,q4,q2),
  app(q2,q3,q3),app(q2,q2,q2),app(q2,q1,q3),
  app(q1,q6,q5),app(q1,q5,q5),app(q1,q4,q1),
  app(q1,q3,q3),app(q1,q2,q3),app(q1,q1,q1),
  app(q4,A,A)}

  {rev(q4,q4),rev(q3,q3),rev(q2,q2),rev(q1,q1)}
  Compact representations are essential!

Infinite State Model Checking

Prolog program representing operations on a token ring (with any number of processes) (example from Roychoudhury et al.).

```
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)}

Is it practical?

- Analysis of a program based on and FTA presents two significant practical challenges
  - Determinization can cause a blow-up in the number of states and transitions
  - Representation and manipulation of relations as tuples is expensive
    - it is like representing Boolean functions using truth tables.

Approaches to Scaling up

- Determinization.
  - Product form of transitions yields much more compact representation of DFTAs
  - Representation of relations. Use a BDD-based representation and exploit techniques from model-checking
  - But of course there is no escape from exponential worst case complexity, so we may need to make further approximations

Product representation of transitions

- $f(Q1,\ldots,Qn) \rightarrow q$ represents the set of transitions

  $\{f(q1,\ldots,qn) \rightarrow q \mid qj \in Qj,\ 1 \leq j \leq n\}$

  E.g. determinized list/nonlist example

  [] → list
  [{list,nonlist}|{list}] → list
  [{list,nonlist}|{nonlist}] → nonlist
  f({list,nonlist},..., {list,nonlist}) → nonlist

Determinization algorithm generating product form

$\mathrm{qmap}(q, f^n, j) = \{f(q_1, \ldots, q_n) \rightarrow q_0 \in \Delta \mid j \leq n, q = q_j\}$
$\mathrm{Qmap}(Q_0, f^n, j) = \cup\{\mathrm{qmap}(q, f^n, j) \mid q \in Q_0\}$
$\mathrm{states}(\Delta) = \{q_0 \mid f(q_1, \ldots, q_n) \rightarrow q_0 \in \Delta\}$

$\mathrm{fmap}(f^n, i, D) = \{\mathrm{Qmap}(Q_0, fn, i) \mid i \leq n, Q_0 \in D\} \setminus \emptyset$

$C = \{q \mid f^0 \rightarrow q \in \Delta\} \mid f^0 \in \Sigma\}$

$F(D) = (\{\mathrm{states}(\Delta_1 \cap \cdots \cap \Delta_n) \mid \Delta_i \in \mathrm{fmap}(f^n, i, D), 1 \leq i \leq n\} \setminus \emptyset) \cup C$

The algorithm finds the least set $D \in 2^{2^D}$ such that $D = F(D)$.
The set D is computed by a fixpoint iteration as follows.

initialise $i = 0$; $D_0 = \emptyset$; repeat $D_{i+1} = F(D_i)$; $i = i + 1$ until $D_i = D_{i-1}$

Example: list/nonlist

t1: [] → list,
t2:[dynamic|list] → list,
t3: [] → dynamic,
t4: [dynamic|dynamic] → dynamic,
t5: f(dynamic,dynamic) → dynamic,
. . .
qmap(list,cons,1) = {}
qmap(list,cons,2) = {t2}
qmap(list,f,1) = {}
qmap(list,f,2) = {}
qmap(dynamic,cons,1) = {t2,t4}
qmap(dynamic,cons,2) = {t4}
qmap(dynamic,f,1) = {t5}
qmap(dynamic,f,2) = {t5}

Example: continued

- D0 = ∅
- D1 = {{t1,t3}}
- 2nd iteration
  - fmap(cons,1,D1) = fmap(cons,2,D1) = {{t2,t4}}
  - fmap(f,1,D1) = fmap(f,2,D1) = {{t5}}
  - D2 = F(D1) = {{t1,t3},{t2,t4},{t5}}
- 3rd iteration
  - fmap(cons,1,D2) = {{t2,t4}}
  - fmap(cons,2,D2) = {{t2,t4},{t4}}
  - fmap(f,1,D2) = fmap(f,2,D2) = {{t5}}
  - D3 = F(D2) = {{t1,t3},{t2,t4},{t5},{t4}}
- D4=D3

Extracting product transitions

fmap(cons,1,D3)	fmap(cons,2,D3)
{{t2,t4}}	{{t2,t4} ,{t4}}

To generate the product transitions for cons, form the product of the fmap values.

[{t2,t4}|{t2,t4}] → {t2,t4}∩{t2,t4}
[{t2,t4}|{t4}] → {t2,t4}∩{t4}

[{{list,dynamic},{dynamic}}|{{list,dynamic}}] → {list,dynamic}
[{{list,dynamic},{dynamic}}|{{dynamic}}] → {dynamic}

Reduction in size with product representation

Q	Δ	Q_d	(Δ_d)	Δ_Π
3	1933	4	(1130118)	1951
4	1934	5	(10054302)	1951
3	655	4	(20067)	433
4	656	5	(86803)	433
105	803	46	(6567)	141
16	65	16	(268436271)	89

Q = no. of FTA states
Δ = no. of FTA rules
Q_d = no. of DFTA states
Δ_d = no. of DFTA rules
Δ_Π = no. of DFTA product rules

Some more results

	Q	Δ	Q_d	Δ_d	Δ_p	Δ_{dc}
chr	21	64	46	118837	242	86
dnf	104	791	57	6567	168	141
mat1	6	10	8	39	8	8
mat2	3	8	3	12	9	7
ring	5	12	5	30	14	11
pic	8	270	8	4989	274	280

Q=original states
Δ=original transitions
Q_d =determinized states
Δ_d = determinized transitions
Δ_p = product transitions
Δ_d = product transitions with *don't cares*

BDD-representation of relations

- Let R be a relation in D^n where D is a finite set with m elements.
- Code the m elements using $k = \lceil \log_2(m) \rceil$ bits each
- introduce n.k Boolean variables $x_{1,1}, \ldots, x_{1,k}, x_{2,1}, \ldots, x_{n,1}, \ldots, x_{n,k}$.
- A tuple in R is then a conjunction
  $x_{1,1} = b_{1,1} \wedge\vee \ldots \wedge, x_{n,k} = b_{n,k}$
  where $b_{i,1} \cdots b_{i,k}$ is the encoding of the i^{th} component of the tuple.
- The whole relation is a disjunction of such conjunctions.

Using Binary Decision Diagrams

- A BDD is a representation of a Boolean function as a graph or decision tree.
- It can give much more compact representations of some large Boolean functions.
- BDDs are successfully used in verification of hardware (since a digital circuit can be represented as a (large) Boolean function)

Example: Relations as Boolean functions

- E.g. let $R \subseteq D^n$ where D={a,b,c,d}
- Let relation R be
  {<a,a,b>, <d,a,b>,<c,d,a>,<b,d,c>}
- How to represent this as a Boolean function?

Mapping to Boolean formulas

- Code the domain elements as bit strings
  - e.g. a = 00, b = 01, c = 10, d = 11
- Introduce one variable per bit
  - e.g. for relation R with 3 arguments, and 2 bits per argument, there are 6 bits
  - x1, x2, x3,...,x6
- Each tuple in the relation is a boolean conjunction of 6 variables (positive = 1, negative = 0)
  - <a,a,b> = ¬x1.¬x2.¬x3.¬x4.¬x5.x6

    0 0 0 0 0 1

Mapping complete relation

- R = {<a,a,b>, <d,a,b>,<c,d,a>,<b,d,c>}
  = ¬x1.¬x2.¬x3.¬x4.¬x5.x6
  + x1.x2.¬x3.¬x4.¬x5.x6
  + x1.¬x2.x3.x4.¬x5.¬x6
  + ¬x1.x2.x3.x4.x5.¬x6
- Relational operations (join, projections etc.) can then be handled using BDD operations
- For our experiments, we use a publicly available BDD package BuDDy
  - ~~http://www.itu.dk/research/buddy~~
  - http://sourceforge.net/projects/buddy
- We also use a relation manipulation package based on BuDDy, called bddbddb
  - ~~http://suif.stanford.edu/bddbddb~~
  - http://bddbddb.sourceforge.net/

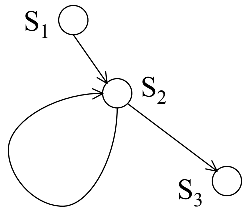
Computing an FTA Approximation of a Programs

Aim of <u>set-based analysis</u> - to find a regular tree approximation of the set of terms 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 a regular tree language)

Need for FTA Analysis for On-line 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	$\rightarrow$ cons(Pcont,S_1) | cons(Rcont,S_2)
S_1	$\rightarrow$ cons(Qcont,Stack)
S_2	$\rightarrow$ emptyStack

Stack = (Pcont Qcont)*Rcont

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

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

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

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

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


Set-Based Analysis

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

Other Variations

- Top-down deterministic DTTAs vs. FTAs
  - precision (FTAs) vs. efficiency (DTTAs)
- Finite height abstract domain vs. infinite height domains
  - with various widening operators
- Constraint solving techniques

Limited Precision of Top-Down Deterministic FTAs

append([], Ys,Ys).
append([X|Xs], Ys, [X|Zs]) $\leftarrow$ append(Xs,Ys,Zs).

?- append(A,B,C).

?

[] $\rightarrow$ A
[a | A] $\rightarrow$ A

[a,a,….a]

[] $\rightarrow$ B
[b | B] $\rightarrow$ B

[b,b,….b]

with a *deterministic* automaton, the best we can do is
[] $\rightarrow$ C
[D | C] $\rightarrow$ C
a $\rightarrow$ D
b $\rightarrow$ 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

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

Concrete Semantics

- The concrete domain consists of sets of relations (atomic formulas)
- The concrete semantic function T performs "one forward inference step"

T(X) = Y, where X and Y are sets of atomic formulas.
To evaluate T(X) for each program clause $H \leftarrow B_1,\ldots,B_n$

1. Solve the body $B_1,\ldots,B_n$ in the set of atomic formulas X yielding a set of substitutions for variables in $B_1,\ldots,B_n$.
2. Project the substitutions onto the head H.

Abstract Semantics

- The abstract domain consists of the set of all NFTAs over a fixed (program-specific) set of states and functions.
  - *The concretisation function*

  $\gamma(A) = L(A)$ (the language of the NFTA A)
  - *The domain ordering*

  $<Q,q^*,\Delta_1> \;\leq\; <Q,q^*,\Delta_2>$ if $\Delta_1 \subseteq \Delta_2$

  (I.e. not the language ordering, but subset ordering on the set of rules).

Abstract Semantics

Define the abstract semantic function S

S(X) = Y where X and Y are NFTAs

For each clause H ← B1,…,Bn
1. solve the body B1,…,Bn wrt X yielding a set of automata states describing the values of variables in B1,…,Bn.
2. project onto H, yielding transitions in Y

Defining automata states

- Given a program, define the set of points which we observe.
- Associate an automaton state with each point.
  - several points may be associated to a single state.
  - there are various possible ways to associate states to points, resulting in different precision in the analysis.

reverse([], []).
reverse([X|Xs],Ys) ← . . .

Associate a distinct state with *each head position*,
OR
associate a state to *each argument.*

Head transitions

reverse([], []).
reverse([X|Xs],Ys) ← . . .

Arg abstraction
reverse(r1,r2) → type
[] → r1
[] → r2
[q1 | q2] → r1
q3 → r2

where q1, q2, q3 are states associated with X, Xs, Ys

Var abstraction
reverse(q4, q5) → type
reverse(q6, q7) → type
[] → q4
[] → q5
[q1 | q2] → q6
q3 → q7

where q1, q2, q3 are states associated with X, Xs, Ys
q4, q5, q6, q7, are associated with terms in the heads.

Solving the clause body

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

Given an NFTA, say R, compute the state(s) that represent the values in R for each body variable. Example:

Xs : r1
Zs: r2
Zs: a1
X: any
Ys: a3

For each variable that occurs more than once, check that the intersection of the corresponding states is non-empty.

Show that r2 ∩ a1 (a product automaton) is non-empty.
Emptiness is decidable for NFTAs.

Projection

rev([X|Xs],Ys) ← rev(Xs,Zs), append(Zs,[X],Ys).
q2 q3 r1 a3

r1→ q2 (an *epsilon transition* which can be eliminated and replaced by a set of ordinary transitions)

a3 → q3 similarly

Example: naïve reverse

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

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

Result for append:
append(a1,a2,a3) → type
[] → a1
[any | a1] → a1
any → a2
any → a3

Iterations for reverse:
1. reverse(r1,r2) → type — S(⊥)
   [] → r1, [] → r2
2. [q1 | q2] → r1 — $S^2(\perp)$
   any → q1
   [] → q2, any → r2
3. [q1 | q2] → q2 — $S^3(\perp)$ $= S^4(\perp)$

Termination

- The analysis terminates
  - least fixed point of S is found after a finite number of iterations, because
  - the set of NFTAs for a given program is finite, since the number of states is finite, and the signature is finite.
    - hence the set of possible transitions is finite
  - each iteration simply adds transitions to the NFTA until no more can be added.
    - can also "grey out" subsumed transitions
    - "greyed out" transitions are not used further except to check against added transitions.

Subsumed transitions

- A transition t = f(q1,…,qn) → q_0 is subsumed by a set of transitions Δ if <Q,q_0,Δ> = <Q,q_0,Δ∪{t}>

A full subsumption check is expensive but we can easily detect some cases, especially where the special state any occurs.

```
f(any) → q
f(q1) → q
f(q2) → q
```

⇒

```
f(any) → q
f(q1) → q
f(q2) → q
```

subsumed transitions *greyed out.*

Tabulation of Non-Empty Product Automata

- Checking non-emptiness of intersection states (product automata) can be expensive.
- Automata grow monotonically
  - once a product (q1 ∩ q2) has been shown to be non-empty, it remains non-empty.
  - *….even though the definitions of q1 and q2 change*
- Hence, we tabulate the non-empty products
  - *never recheck emptiness of the same product*

Experiments

- See PADL'02 paper for experimental results and some aspects of the implementation.
- Results compare favourably with set-based analysis
  - more experiments needed
- Precision compares favourably with deterministic types obtained by abstract interpretation.
- Larger programs handled than previous methods (4000+ clauses of Prolog).

Specialization Examples

- Unification algorithm specialized for ground terms, reduces to term identity.
  - the set of ground terms is represented as an NFTA
- Specialization of regular parsers w.r.t. given regular expressions.
- Specialization as "infinite state model checking".

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.
```

Infinite State Model Checking

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

```
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).
bad([0|X]) ← bad(X).
bad([1|X]) ← one(X).
one([0|X]) ← one(X).
one([1|X]).
unsafe(X) ← reachable(X), bad(X).
```


Adding constraints

- Represent transitions as regular unary logic clauses

f(q1,…,qn) → q0 represented as

q0(f(x1,…,xn)) ← q1(x1),…,qn(xn)

Add a constraint on the variables of the clause

q0(f(x1,…,xn)) ← c(x1,…,xn), q1(x1),…,qn(xn)

- We consider linear arithmetic constraints, and equality/disequality constraints.

Extensions preserving decidability

- If c(X1,...,Xn) consists of equalities, disequalities, and arithmetic inequalities, the emptiness problem remains decidable.
- If we extend to allow equalities, disequalities, and arithmetic inequalities between terms at different level then we lose decidability.
  - E.g. we can represent classic undecidable problems like the Post correspondence problem using such a language.

Example: constrained transitions

- A sorted list of positive numbers

```
sorted(X1) ←t1(X1).
t1([]) ←true.
t1([X1|X2]) ←X2=[],X1>=0,
        any(X1),t2(X2).
t1([X1|X2]) ←X2=[X3|X4], X1>=0, X1-X3>=0,
        any(X1),t1(X2).
t2([]) ←true.
```

But - emptiness of NFTAs with arbitrary constraints is not decidable!

A pragmatic solution

Implement a partial non-emptiness check.

We do not know whether the results of the analysis are empty or not.

(but results are strictly more precise than ordinary NFTAs)

Approaches Using Widening

- Consider a domains of FTAs with an unlimited supply of states.
- There is an infinite set of FTAs that can be constructed, and infinite chains of FTAs ordered by language inclusion.

- Abstract interpretation over such a domain requires a widening operation in order to terminate.

Example. Widening FTAs

- Iterations of T_{append}
  1. {append([],X,X)}
  2. {append([],X,X), append([A],X, [A|X])}
  3. {append([],X,X), append([A],X, [A|X]), append([A,B],X, [A,B|X]), }
  4. · · ·
- The successive terms can be described reasonably accurately by the following sequence of FTAs.
  1. R1 = {append(q1, any, any) → type, [] → q1}
  2. R2 = R1 [{append(q2, any, q3) → type, [any|q1] → q2, [] → q1, [any|any] → q3}
  3. R3 = R2 [{append(q4, any, q5) → type, [any|q2] → q4, [any|q1] → q2, [] → q1, [any|any] → q3, [any|q3] → q5}
  4. · · ·

Introducing recursive transitions

- It can be seen that this sequence could be continued indefinitely
- each iteration extends the terms accepted by the first argument of append.
- There are various widening methods which would “notice” the growth of the first argument and introduce a recursive transition which is a fixpoint.
- 5. R4 = {append(q6, any, q3) → type,[]→ q6, [any|q6] → q6, [any|any] → q3}

Tradeoffs

- The tradeoffs of precision and complexity are not completely understood.
- FTAs vs. DTTAs
  - when to approximate an FTA by a DTTA?
- Different widenings
- Delaying widening
- Whether to use DFTAs and DFTA minimization algorithms (not covered in these lectures) rather than NFTAs