Abstract Interpretation Based Static Analysis Parameterized by Semantics

Patrick Cousot

École normale supérieure, DMI
45 rue d'Ulm
75230 Paris cedex 05, France

cousot@dmi.ens.fr

http://www.dmi.ens.fr/~cousot

1977 Objectives

From the introduction of P. Cousot and R. Cousot, POPL'77, p. 238 [CC77]:

> The conclusion points out that abstract interpretation of programs is a unified approach to apparently unrelated program analysis techniques.

Reference

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In 4^{th} *POPL*, pages 238–252, Los Angeles, Calif., 1977. ACM Press.

1997 Objectives

- The number of apparently unrelated program analysis techniques continues to increase;
- After suitable evolutions, the theory of abstract interpretation is still a unifying framework [1] to explain and justify these program analysis techniques;
- This understanding of abstract interpretation has evolved beyond program analysis as a unifed approach to apparently unrelated semantics;
- We now think of it as an approximate computation model.

Can abstract interpretation become a universal approximate computation model?

[1] most often superbly ignored by the creators of these apparently unrelated program analysis techniques.

Ingredients of an Abstract Interpretation

- A programming/specification language;
- A standard/concrete semantic domain (objects/operations);
- A concrete *semantics* describing computations;
- An approximation specification (abstraction/concretization);
- An *abstract semantic domain* (objects/operations);
- An abstract semantics approximating computations;
- An *abstract interpreter* computing the abstract semantics;
- Applications.

Abstract Interpretation

- Languages: many (imperative, functional, logic, constraint-based, parallel, object-oriented, etc.);
- Concrete semantics: many (operational, denotational, relational, axiomatic, etc.);
- Abstract domains: many, often specialized (strictness analysis, sharing analysis, etc.), often algorithmically involved (polyhedra);
- Abstract interpreters: many, complex, specialized for one language & often one type of program analysis;
- Applications: many, unkown in advance (e.g. programs).

This great diversity makes abstract interpretation very difficult.

Difficulty of Abstract Interpretation

- Abstract interpretation has a very broad scope of application, from practice (compilers) to theory (semantics);
- Abstract interpretation requires competences in many domains:
  - Mathematics & Semantics,
  - Languages & Compilers,
  - Algorithms,
  - Programming skill;
- What about other static analysis methods?

Data Flow Analysis

- Languages: few (program graphs);
- Concrete semantics: none (or informal ones);
- Abstract domains: few (booleans, pointers);
- Abstract interpreters: integrated in a few compilers;
- Applications: many programs, unkown in advance.

Reference

[MR90] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks: A unified model. *Acta Inf.*, 28:121–163, 1990.

Type Inference [Mil78]

- Languages: few (λ-calculus, ML);
- Concrete semantics: few (operational/denotational);
- Abstract domains: few (Herbrand abstract domain);
- Abstract interpreters: integrated in a few compilers;
- Applications: many programs, unkown in advance.

Reference

[Mil78] R. Milner. A theory of polymorphism in programming. *J. Comput. Sys. Sci.*, 17(3):348–375, dec 1978.

Model Checking

- Languages: few (binary hardware models, (timed) abstract processes);
- Concrete semantics: one (transition systems);
- Abstract domains: few (BDDs, polyhedra);
- Abstract interpreters: few specialized (e.g. Concurrency Workbench);
- Applications: very similar, often specialized to a specific example.

Reference

[CES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *10th POPL*. ACM Press, jan 1983.

The specificity and more limited scope of these analysis methods make them more easily understandable hence applicable.

How can we make abstract interpretation more easily applicable?

Universal Abstract Interpretations

- Are there universal *semantics* which can be used as a basis for all program analysis algorithms?
- Are there universal *abstract semantic domains* that can be used for many different program analysis algorithms?
- Are there universal *abstract interpreters* that can be used for many different analyses of many different programming languages?
- How can this be made *efficient*?

Universal Semantics?

No Semantics is General Purpose

- A general purpose abstract interpretation framework can hardly be built on a specific semantics;
- What can be done?
  - Separate the formalization of abstract interpretation from a specific semantics: abstract interpretation can be explained as a theory of approximation in the context of the mathematical structures used by semanticians;
  - Relate semantics by abstraction: all known semantics can be organized in a hierarchy by abstract interpretation.

Sketch of a Hierarchy of Semantics [Cou97]

- Trace operational semantics;
- Transition system operational semantics;
- Relational semantics;
- Non-deterministic denotational semantics;
- Deterministic denotational semantics;
- Predicate transformer semantics:
- Axiomatic semantics

Reference

[Cou97] P. Cousot. Design of semantics by abstract interpretation, invited address. In *Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference (MFPS XIII)*, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 23–26 mar 1997. to appear in ENTCS.

Operational Trace Semantics

$$\mathcal{S} = \{\bullet, \bullet, \ldots\} \qquad \text{states}$$

$$\begin{aligned}\mathcal{T} = & \quad \{\bullet\!\to\bullet\to\ldots\to\bullet\to\!\bullet, \ldots\} & \text{finite traces}\\ & \cup \{\bullet\!\to\bullet\to\ldots\to\bullet\to\bullet\to\ldots, \ldots\} & \text{infinite traces}\end{aligned}$$

The operational trace semantics $\mathcal{T}$ of a transition system has a fixpoint characterization (with respect to a generalization of Scott's ordering) from which fixpoint characterizations of all other semantics can be derived by abstract interpretation.

Transition System Operational Semantics

$$\alpha \in \text{Traces} \longmapsto \wp(\mathcal{S} \times \mathcal{S})$$

$$\begin{aligned}\tau &= \alpha(\mathcal{T})\\ &= \{\langle\bullet, \bullet\rangle \mid \bullet\!\to\ldots\to\!\bullet\!\to\!\bullet\!\to\ldots \in \mathcal{T}\}\end{aligned}$$

Galois connection.

Relational Semantics

$$\alpha \in \text{Traces} \longmapsto \wp(\mathcal{S} \times \mathcal{S}_\perp), \quad \mathcal{S}_\perp = \mathcal{S} \cup \{\perp\}$$

$$\begin{aligned}\mathcal{R} &= \alpha(\mathcal{T})\\ &= \quad \{\langle\bullet, \bullet\rangle \mid \bullet\!\to\bullet\to\ldots\to\bullet\to\!\bullet \in \mathcal{T}\}\\ &\quad \cup \{\langle\bullet, \perp\rangle \mid \bullet\!\to\bullet\to\ldots\to\bullet\to\bullet\to\ldots \in \mathcal{T}\}\end{aligned}$$

Galois connection.

Non-deterministic Denotational Semantics

$$\alpha \in \wp(\mathcal{S} \times \mathcal{S}_\perp) \longmapsto (\mathcal{S} \mapsto \wp(\mathcal{S}_\perp))$$

$$\begin{aligned}\mathcal{D} &= \alpha(\mathcal{R}) \\ &= \lambda s \bullet \{s' \in \mathcal{S}_\perp \mid \langle s, s'\rangle \in \mathcal{R}\} \qquad \text{right image}\end{aligned}$$

α is a Galois isomorphism.

Deterministic Denotational Semantics

$$\alpha \in (\mathcal{S} \mapsto \wp(\mathcal{S}_\perp)) \longmapsto (\mathcal{S} \mapsto \mathcal{S}_\perp^\top), \quad \mathcal{S}_\perp^\top = \mathcal{S} \cup \{\perp, \top\}$$

$$\begin{aligned}\varphi &= \alpha(\mathcal{D}) \\ &= \lambda s \bullet \big(\mathcal{D}(s) \subseteq \{\perp\}\ ?\ \perp \ \big|\ \mathcal{D}(s) \subseteq \{s', \perp\}\ ?\ s'\ ¿\ \top\big)\end{aligned}$$ [2]

Galois connection.

$\top$ can be eliminated for deterministic systems.

[2] $(\ldots ? \ldots \mid \ldots ? \ldots ¿ \ldots)$ is if … then … elsif … then … else …

Predicate Transformer Semantics

$$\alpha \in (\mathcal{S} \mapsto \wp(\mathcal{S}_\perp)) \longmapsto (\wp(\mathcal{S}) \stackrel{\cup}{\longmapsto} \wp(\mathcal{S}_\perp))$$

$$\begin{aligned}\mathcal{W} &= \alpha(\mathcal{D}) \\ &= \lambda P \bullet \{s' \in \mathcal{S}_\perp \mid \exists s \in P : s' \in \mathcal{D}(s)\}\end{aligned}$$ [3]

α is a Galois isomorphism.

Axiomatic Semantics

$$\alpha \in (\wp(\mathcal{S}) \stackrel{\cup}{\longmapsto} \wp(\mathcal{S}_\perp)) \longmapsto (\mathcal{S} \otimes \mathcal{S}_\perp)$$

$$\begin{aligned}\mathcal{H} &= \alpha(\mathcal{W}) \\ &= \{\langle P, Q\rangle \mid \mathcal{W}(P) \subseteq Q\}\end{aligned}$$

α is a Galois isomorphism.

$$\begin{aligned}\mathcal{S} \otimes \mathcal{S}_\perp = \{\mathcal{H} \in \wp\big(\wp(\mathcal{S}) \times \wp(\mathcal{S}_\perp)\big) \mid \\ &(P \subseteq P' \wedge \langle P', Q'\rangle \in \mathcal{H} \wedge Q' \subseteq Q) \Longrightarrow \langle P, Q\rangle \in \mathcal{H} \wedge \\ &(\forall i \in \Delta : \langle P_i, Q\rangle \in \mathcal{H}) \Longrightarrow (\langle \textstyle\bigcup_{i \in \Delta} P_i, Q\rangle \in \mathcal{H}) \wedge \\ &(\forall i \in \Delta : \langle P, Q_i\rangle \in \mathcal{H}) \Longrightarrow (\langle P, \textstyle\bigcap_{i \in \Delta} Q_i\rangle \in \mathcal{H})\end{aligned}$$

[3] Strongest post-condition generalized to handle the possibility of non-termination, denoted $\perp$.

Natural, Demoniac & Angelic Semantics

- Natural trace semantics: $\mathcal{T}^\natural$;
- Angelic abstraction:

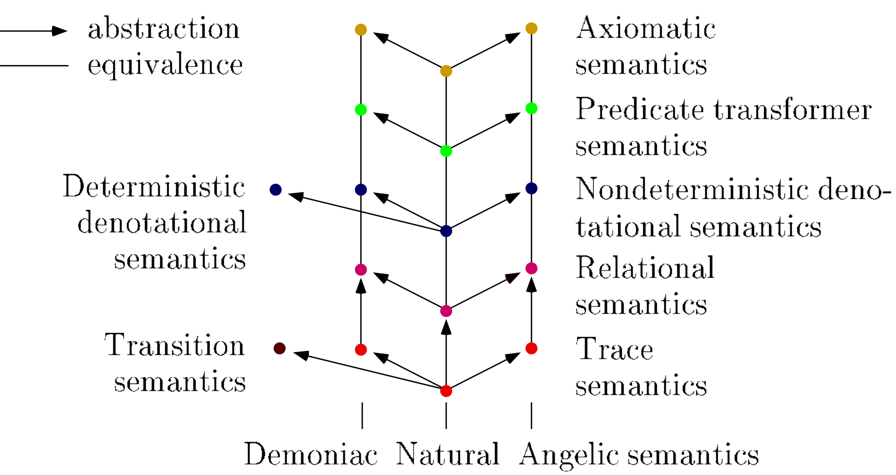
$$\alpha^\flat(\mathcal{T}^\natural) = \{\bullet\!\to\bullet\to\ldots\to\bullet\to\!\bullet \mid \bullet\!\to\bullet\to\ldots\to\bullet\to\!\bullet \in \mathcal{T}^\natural\};$$

- Demoniac abstraction:

$$\alpha^\sharp(\mathcal{T}^\natural) = \mathcal{T}^\natural \cup \{\bullet\!\to\bullet\to\ldots\to\bullet\to\!\bullet \mid \bullet\!\to\bullet\to\ldots\to\bullet\to\bullet\to\ldots \in \mathcal{T}^\natural\}.$$

Galois connections.

The Hierarchy of Semantics



Towards Universal Abstract Semantic Domains

No Abstract Domain is General Purpose

- By the abstraction process, some properties will always be intrinsically unexpressible;
- Expressive abstract domains are algorithmically complex (e.g. polyhedra);
- Expressive power is very difficult to conciliate with efficiency (widening);
- The design of special purpose abstract domains is hard anyway, so why not try very general purpose ones?
- The relative success of the few large scope abstract domains is stimulating!

Requirements

- Can be used to handle sets of numerical values;
- Can be used to handle sets of complex data structures (vectors, lists, trees, graphs, ...);
- Can be used to express control structures (functions, relations, ...);
- Abstract operations can be used to formulate abstract semantics;
- [Can be used to express/approximate concrete semantics.]

Origins

- Sets of terms for analyzing pure Lisp programs by J. Reynolds [Rey69];
- Formalization using deterministic tree grammars & projection elimination algorithm by Jones and Muchnich [JM79];
- Rephrased and popularized as set constraints and set-based analysis by N. Heinze [Hei92];

Reference

Rey69] J. Reynolds. Automatic computation of data set definitions. In *Information Processing 68*. North-Holland, 1969.

JM79] N.D. Jones and S.S. Muchnich. Flow analysis and optimization of LISP-like structures. In *6th POPL*, pages 244–256, San Antonio, Texas, 1979. ACM Press.

Hei92] N. Heintze. *Set Based Program Analysis*. PhD, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1992.

- Shown to be an abstract interpretation with finite domain where constraint resolution is isomorphic to a chaotic iterative fixpoint computation by P. Cousot & R. Cousot [CC95];
- Proposition to enrich into deterministic constrained tree grammars (using a numerical domain to count derivations) by P. Cousot & R. Cousot [CC95];
- Algorithm design and implementation (using M. Karr affine equality relationships [Kar76]) by A. Venet [Ven97].

References

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. 7th FPCA*, pages 170–181, jun 1995. ACM Press.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[Ven97] A. Venet. Program analysis using context-sensitive grammars. Manuscript, LIX, École Polytechnique, Palaiseau, FRA, 1997.

The Data Set Definition Example of J. Reynolds

Given the recursive definition of its argument $\mathtt{x1} \in X_1$:

$$X_1 = \mathtt{nil} \cup \mathtt{cons}(\underline{\mathtt{atom}}, X_1),$$

the function `ss(x1)` accepts a list `x1` representing a set of atoms and produces a list of lists representing all subsets of the set of atoms:

```
           2                 3    4   4
ss(x1) = if null(x1) then cons(nil,nil)
              5   6   7   8  9   7
           else ss1(car(x1),ss(cdr(x1)))
             10  11  12  12
ss1(x2,x3) = ss2(x2, x3, x3)
                 13                14
ss2(x4,x5,x6) = if null(x5) then x6
                     15   16   17 18  19   20    21  19  14
                else cons(cons(x4,car(x5)),ss2(x4,cdr(x5),x6))
```

- Recursive set definition which is a “good fit” to the results of the labeled expressions:

$$
\begin{array}{llll}
X_2 \leftarrow X_3 \cup X_5 & & X_{12} \leftarrow X_8 \\
X_3 \leftarrow \mathtt{cons}(X_4, X_4) & & X_{13} \leftarrow X_{14} \cup X_{15} \\
X_4 \leftarrow \{\mathtt{nil}\} & & X_{14} \leftarrow X_{12} \cup X_{14} \\
X_5 \leftarrow X_{10} & & X_{15} \leftarrow \mathtt{cons}(X_{16}, X_{20}) \\
X_6 \leftarrow \mathtt{car}(X_7) & & X_{16} \leftarrow \mathtt{cons}(X_{17}, X_{18}) \\
X_7 \leftarrow X_9 \cup X_1 & & X_{17} \leftarrow X_{11} \cup X_{17} \\
X_8 \leftarrow X_2 & & X_{18} \leftarrow \mathtt{car}(X_{19}) \\
X_9 \leftarrow \mathtt{cdr}(X_7) & & X_{19} \leftarrow X_{12} \cup X_{21} \\
X_{10} \leftarrow X_{13} & & X_{20} \leftarrow X_{13} \\
X_{11} \leftarrow X_6 & & X_{21} \leftarrow \mathtt{cdr}(X_{19})
\end{array}
$$

- The equations are ***simplified*** by elimination of `car` and `cdr` using LISP identities, like:

$$
\begin{aligned}
\mathtt{car}(\mathtt{cons}(X, Y)) &= X, \\
\mathtt{cdr}(\mathtt{cons}(X, Y)) &= Y, \\
\mathtt{car}(\underline{\mathtt{atom}}) &= \emptyset, \ldots
\end{aligned}
$$

- However the equations are not ***solved*** in some normal form.

- J. Reynolds’ results:

$$
\begin{array}{rl}
X_1, X_7, X_9 \leftarrow & \mathtt{nil} \cup \mathtt{cons}(\mathtt{atom}, X_1) \\
X_2,\ X_5,\ X_8,\ X_{10},\ X_{12},\ X_{13},\ X_{14},\ X_{20} \leftarrow & \mathtt{cons}(X_4, X_4) \cup \mathtt{cons}(X_{16}, X_{20}) \\
X_3 \leftarrow & \mathtt{cons}(X_4, X_4) \\
X_4 \leftarrow & \mathtt{nil} \\
X_6, X_{11}, X_{17} \leftarrow & \underline{\mathtt{atom}} \\
X_{15} \leftarrow & \mathtt{cons}(X_{16}, X_{20}) \\
X_{16} \leftarrow & \mathtt{cons}(X_{17}, X_{18}) \\
X_{18} \leftarrow & \mathtt{nil} \cup \mathtt{cons}(X_{17}, X_{18}) \\
X_{19}, X_{21} \leftarrow & \mathtt{nil} \cup \mathtt{cons}(X_4, X_4) \cup \mathtt{cons}(X_{16}, X_{20})
\end{array}
$$

- X_2, which must include all results of `ss`, is the set of all non-empty lists whose last element is `nil` and whose preceding elements are non-empty lists of atoms.

- Equivalent formulation of the equations ($\in$ is written `<`):

```
widen(X1, X8, X20).

X1  = {nil} + {cons(atom,T) | T < X1}.
X2  = {X | X < X3} + {X | X < X5}.
X3  = {cons(X,X) | X < X4}.
X4  = {nil}.
X5  = {X | X < X10}.
X6  = {X | cons(X, Y) < X7}.
X7  = {X | X < X9} + {X | X < X1}.
X8  = {X | X < X2}.
X9  = {Y | cons(X, Y) < X7}.
X10 = {X | X < X13}.
X11 = {X | X < X6}.
X12 = {X | X < X8}.
X13 = {X | X < X14} + {X | X < X15}.
X14 = {X | X < X12} + {X | X < X14}.
X15 = {cons(X,Y) |
               X < X16, Y < X20}.
X16 = {cons(X,Y) |
               X < X17, Y < X18}.
X17 =   {X | X < X11}
      + {X | X < X17}.
X18 = {X | cons(X, Y) < X19}.
X19 =   {X | X < X12}
      + { X | X < X21}.
X20 = {X | X < X13}.
X21 = {Y | cons(X, Y) < X19}.
```

The Regular Tree Grammar Example of N. Jones and S. Muchnich

- The program builds a linear tree X from input items, and then transfers them to Y in their original order:

```
{0} repeat
{1}    readint(Z);
       X := cons(Z,X);
{2} until (Z = 0);
{4} while (X <> nil) do
{5}    Y := cons(hd(X),Y);
       X := tl(X);
{6} od;
{7}
```


- Equations:

```
I0 = {i(nil,nil,nil)}.
I1 =   {i(X,Y,Z) | i(X,Y,Z) < I0}
     + {i(X,Y,Z) | i(X,Y,Z) < I2}.
I2 = {i(cons(atom,X1),Y1,atom) | i(X1,Y1,Z1) < I1}.
I4 = {i(X,Y,atom) | i(X,Y,atom) < I2}.
I5 =   {i(X,Y,Z) | i(X,Y,Z) < I4}
     + {i(X,Y,Z) | i(X,Y,Z) < I6}.
I6 = {i(Xtl,cons(Xhd,Y),Z) | i(cons(Xhd,Xtl),Y,Z) < I5}.
I7 = {i(nil,Y,Z) | i(X,Y,Z) < I4}
   + {i(nil,Y,Z) | i(X,Y,Z) < I6}.
```


- Solution in normal form: regular tree grammars:

```
...

Solution of variable I7 :
-------------------------

S -> i(X,Y,Z)
X -> nil
Y -> cons(A,Y)
Y -> nil
Z -> atom
A -> atom
```


A. Aiken & N. Heintze set based analysis

- N. Heintze extended N. Jones and S. Muchnich algorithms (intersection);
- With A. Aiken, he provided numerous examples of analysis and typing of imperative, logic, higher-order functional languages;
- Clearly explained the limitations of set-based analysis;
- Claimed that set-based analysis is *not* an abstract interpretation, even using widening.

Set-Based Analysis is an Abstract Interpretation [CC95]

- The (erroneous) counterexample given by A. Aiken & N. Heinze [AH95] is:

```
X := cons(a, nil);
{a}
while (X <> nil) do
  {b}
  X := cons(a, X);
  {c}
{d}
```

Reference

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc.* 7^{th} *FPCA*, pages 170–181, La Jolla, Calif., 25–28 jun 1995. ACM Press.

[AH95] A. Aiken and N. Heinze. Invited talk. POPL'95, 1997.

- Equations (equivalent to set constraints by Tarski's fixpoint theorem):

```
IsNil = {nil}.
Xa    = {cons(a, nil)}.
Xb    = {X | X < Xa} + {X | X < Xc}.
Xc    = {cons(a, X) | X < Xb}.
Xd    =   {X | X < Xa, X < IsNil}
        + {X | X < Xc, X < IsNil}.
```

- For a given program, the abstract semantic domain (regular tree grammars with a finite alphabet) is *finite*;
- For the counter-example, N. Heinze uses a different *infinite* abstract domain (arbitrary sets of arbitrary trees). So the comparison using different abstract domains is unfair!

- Chaotic iterative resolution (isomorphic to constraint resolution) with appropriate abstract domain exactly provides the expected solution:

```
Solution of variable Xd :
-------------------------

Empty Grammar

Solution of variable Xc :
-------------------------

S -> cons(A,B)
A -> a
B -> cons(C,B)
B -> nil
C -> a
```


Limitations of Set-Based/Regular Tree Grammar Analysis

- Essentially non-relational, grammars cannot express context conditions [HJ90]:

```
p(a,b).
p(b,a).
q(X,Y) <-- p(X,Y).

Psem = {p(a,b)} + {p(b,a)}.
Qsem = {q(X,Y) | p(X,Y) < Psem}.
```

Reference

[HJ90] N. Heintze and J. Jaffar. A Finite Presentation Theorem for Approximating Logic Programs. In 17^{th} *POPL*, pages 197–209, 1990. ACM Press.

```
Solution of variable Psem :
---------------------------

- Grammar

S -> q(A,B)
A -> b
A -> a
B -> b
B -> a
```

```
Solution of variable Qsem :
---------------------------

- Grammar

S -> p(A,B)
A -> b
A -> a
B -> b
B -> a
```

- Does not exclude the impossible cases `p(a,a)` and `p(b,b)`!

Constrainted Tree Grammars [CC95]

- Add counters to count the number of times each production in the grammar can be used in a derivation;
- Express the possible values of the counters using numerical constraints [Deu92];

Reference

[CC95] P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proc. 7th FPCA*, pages 170–181, jun 1995. ACM Press.

[Deu92] A. Deutsch. A storeless model of aliasing and its abstraction using finite representations of right-regular equivalence relations. In *Proc. 1992 ICCL, Oakland, Calif.*, pages 2–13. IEEE Comp. Soc. Press, 20–23 apr 1992.

Expressing Context Conditions

- Example showing that constrainted tree grammars analysis is strictly more precise than set based analysis [CC95]:

```
P(0, 0, 0).
P(a(X), b(Y), c(Z)) <-- P(X, Y, Z).

widen(P).
P = {p(z, z, z)} +
    {p(a(X), b(Y), c(Z)) | p(X, Y, Z) < P}.
```



```
Solution of variable P :
------------------------

- Grammar                - Constraints

S -> p(A,B,C)            C.z = 1
A -> z                   B.z = 1
A -> a(A)                A.z = 1
B -> b(B)                S.p = 1
B -> z                   B.b-C.c = 0
C -> z                   A.a-C.c = 0
C -> c(C)
```

- $\{\mathtt{p(a}^n\mathtt{(z),b}^n\mathtt{(z),c}^n\mathtt{(z))} \mid n \geq 0\}$ instead of set based analysis $\{\mathtt{p(a}^k\mathtt{(z),b}^\ell\mathtt{(z),c}^m\mathtt{(z))} \mid k \geq 0,\ \ell \geq 0,\ m \geq 0\}$!

Expressing Relations

- Back to the N. Heintze and J. Jaffar example [HJ90]:

```
p(a,b).
p(b,a).
q(X,Y) <-- p(X,Y).

Psem = {p(a,b)} + {p(b,a)}.
Qsem = {q(X,Y) | p(X,Y) < Psem}.
```

- The constraints now exclude the impossible cases `p(a,a)` and `p(b,b)`!

Reference

[HJ90] N. Heintze and J. Jaffar. A Finite Presentation Theorem for Approximating Logic Programs. In *17th POPL*, pages 197–209, 1990. ACM Press.


```
Solution of variable Psem :
---------------------------
- Grammar
    S -> q(A,B)
    A -> b
    A -> a
    B -> b
    B -> a
- Constraints
    S.q = 1
    B.a+B.b = 1
(1) A.a-B.b = 0
(2) A.b+B.b = 1
```

```
Solution of variable Qsem :
---------------------------
- Grammar
    S -> p(A,B)
    A -> b
    A -> a
    B -> b
    B -> a
- Constraints
    S.p = 1
    A.a+A.b = 1
    B.a-A.b = 0
    B.b+A.b = 1
```

(1) If `A` is `a` then `B` is `b`;

(2) Either `A` or `B` is `b`.

Limitations of Constrainted Tree Grammars

- *Good* at approximating *infinite* sets of trees;
- *Poor* at representing exactly *finite* sets of trees!

Finite Set Counter-Example

```
Or =   {or(tt,tt,tt)} + {or(tt,ff,tt)}
     + {or(ff,tt,tt)} + {or(ff,ff,ff)}.

Orttt = {R | or(tt,tt,R) < Or}.

Solution of variable Orttt :
----------------------------
- Grammar
    S -> ff
    S -> tt
- Constraints
    S.tt+S.ff = 1
```

Finite and Infinite Sets

- The domain must be enriched to combine:
  - An exact representation of finite sets of trees,
  - A finite approximate representation of infinite sets of trees,
  - A widening of large finite sets into approximate infinite supersets.

Exact Representation of Finite Sets

```
tables(Or).

Or =   {or(tt,tt,tt)} + {or(tt,ff,tt)} + {or(ff,tt,tt)}
     + {or(ff,ff,ff)}.

Orttt = {R | or(tt,tt,R) < Or}.

Solution of variable Orttt :
-----------------------------
- Grammar
     S -> tt
- Constraints
     S.tt = 1
```


Implementation of Constrainted Tree Grammars

- Implementation of A. Venet [Ven97]:
  - Deterministic tree grammars with M. Karr's linear equality relationships on derivation counters & exact finite sets of trees;
  - The separate implementation of these domains is easy. Their combination is algorithmically difficult (e.g. test for emptiness).
  - A necessary (parameterized) widening has also been implemented.

Reference

[Ven97] A. Venet. Program analysis using context-sensitive grammars. Manuscript, LIX, École Polytechnique, Palaiseau, 1997.

Abstract values & transformers: set-theoretic style

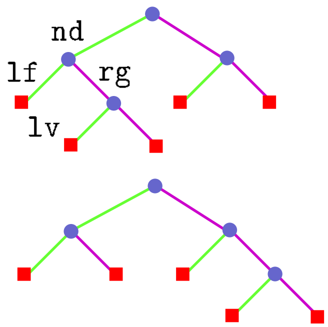
```
widen(BinaryTree).

BinaryTree = {lv} + {nd(X, Y) | X < BinaryTree, Y < BinaryTree}.

Solution of variable BinaryTree :
---------------------------------
- Grammar
     S -> nd(S,S)
     S -> lv
- Constraints
     S.lv-S.nd = 1
```

A well-known theorem on binary trees!

Abstract values: grammar style

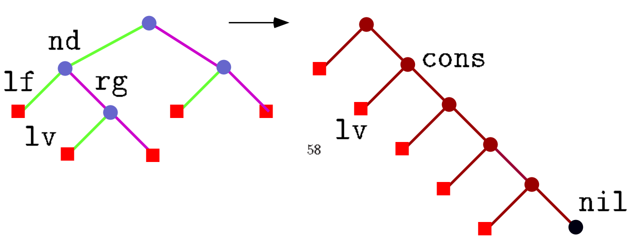
Example: symmetric binary trees, with 5 leaves:

```
#def Tree5leaves = [
  S -> lv;
  S -> nd(L,R);
  L -> lf(S);
  R -> rg(S);
] & [
  S.lv - S.nd = 1;
  L.lf - R.rg = 0;
  2 S.nd - L.lf - R.rg = 1;
  S.lv = 5;
]
```

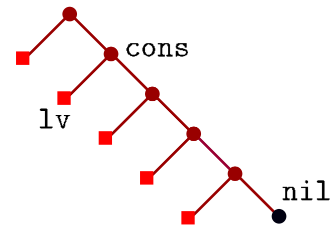


Abstract value transformer: grammar style

```
#fun flatten = [  -> [                 & [
  S -> lv;              S -> cons(H,S);    S.lv - S.nd = 1;
  S -> nd(L,R);         S -> nil;          L.lf - R.rg = 0;
  L -> lf(S);           H -> lv; ]         2 S.nd - L.lf - R.rg = 1;
  R -> rg(S); ]                            S'.nil = 1;
                                           S'.cons - H'.lv = 0;
                                           S.lv - H'.lv = 0; ]
```




```
List = flatten(Tree5leaves) .

Solution of variable List :
---------------------------
- Grammar
     S -> cons(A,S)
     S -> nil
     A -> lv
- Constraints
     S.cons = 5
     S.nil = 1
     A.lv = 5
```



The number of elements in the list is equal to the number of leaves of the tree.

A Few Examples Which Can Be Handled Using Universal Abstract Semantic Domains

Parity Analysis

```
tables(Mult,Minus).

Mult = {mult(odd,odd,odd)} + {mult(odd,even,even)} +
       {mult(even,odd,even)} + {mult(even,even,even)}.

Minus = {minus(odd,odd,even)} + {minus(odd,even,odd)} +
        {minus(even,odd,odd)} + {minus(even,even,even)}.
```


- Program to be analysed:

```
     X := 1; N := 10;
 {1} while (N<>0) do
 {2}   X := X * 2; N := N - 1;
 {3} od;
 {4}
```

- Equations:

```
I1 = {xn(odd,even)}.
I2 = {X | X < I1} + {X | X < I3}.
I3 = {xn(X2,N2) | xn(X1,N1) < I2,
                  mult(X1,even,X2) < Mult,
                  minus(N1,odd,N2) < Minus}.
I4 =   {xn(even,N) | xn(even,N) < I1}
     + {xn(even,N) | xn(even,N) < I3}.
```



```
Solution of variable I3 :
-------------------------

- Grammar

S -> xn(A,B)
A -> even
B -> odd
B -> even

- Constraints

B.even+B.odd = 1
A.even = 1
S.xn = 1
```


Numerical Analysis

```
widen(int,addition).

int = {zero} + {succ(X) | X < int}.
addition = {add(zero, X, X) | X < int} +
           {add(succ(X), Y, succ(Z)) | add(X, Y, Z) < addition,
                                       X < int, Y < int, Z < int}.
Solution of variable addition :
-------------------------------
- Grammar             - Constraints
S -> add(A,B,C)       B.succ+A.succ-C.succ = 0
A -> zero             A.zero = 1
A -> succ(A)          C.zero = 1
B -> zero             B.zero = 1
B -> succ(B)          S.add = 1
C -> zero
C -> succ(C)
```

Boolean Algebra

```
tables(Boolean,Or,And,Iterates).

Boolean = {tt} + {ff}.

Or =   {or(tt, B, tt) | B < Boolean}
     + {or(B, tt, tt) | B < Boolean}
     + {or(ff, ff, ff)}.

And =   {and(ff, B, ff) | B < Boolean}
      + {and(B, ff, ff) | B < Boolean}
      + {and(tt, tt, tt)}.
```


Strictness Analysis [Myc81]

$$f(x,y) = \text{if } x = 0 \text{ then } y \text{ else } 1 + f(x-1,y);$$
$$f^\sharp(X,Y) = X \wedge (Y \vee f^\sharp(X,Y)) \ .$$

```
Iterates =   {F(X, Y, ff) | X < Boolean, Y < Boolean}
           + {F(X, Y, R) | X < Boolean, Y < Boolean,
                 and(X, S, R) < And,
                  or(Y, T, S) < Or,
                   F(X, Y, T) < Iterates}.

NotStrict = {ns(X, Y) | F(X, Y, tt) < Iterates}.
```

Reference

[Myc81] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* Ph.D. Dissertation, CST-15-81, Department of Computer Science, University of Edinburgh, dec 1981.


```
Solution of variable NotStrict :
--------------------------------

- Grammar

S -> ns(A,B)
A -> tt
B -> tt

- Constraints

S.ns = 1
A.tt = 1
B.tt = 1
```


Collecting Semantics of Prolog programs

- Evenness of natural numbers:

```
p(X,Z) :- q(X,Y), r(Y,Z).
q(s(s(X)), Y) :- q(X,Y).
q(z,e).
q(s(z),o).
r(e,yes).
r(o,no).
```

- Syntax of terms:

```
Var   = {w} + {x} + {y}.
Term  = {X | X < Var} + {z} + {s(N) | N < Term} + {o} + {e}
        + {yes} + {no}.
```

- Set Up of terms possibly unifying with parameter p:

```
UssX  = {Y | Y < Var} + {s(Y) | Y < Var} + {s(s(T)) | T < Term}.
```

The term s(s(X)) is unifiable with any term of the form Y, s(Y) and s(s(t)) where Y is a variable and t is any term. Similarly:

```
Uz    = {X | X < Var} + {z}.
Ue    = {X | X < Var} + {e}.
Uo    = {X | X < Var} + {o}.
Usz   = {s(z)} + {s(X) | X < Var} + {X | X < Var}.
Uyes  = {X | X < Var} + {yes}.
Uno   = {X | X < Var} + {no}.
```


- Collecting semantics of the program:

  - p(X,Z) :- q(X,Y), r(Y,Z).:

```
Callp = {p(X,Z) | p(X,Z) < Goal}.
Retp  = {p(X,Z) | q(X,Y) < Retq, r(Y,Z) < Retr}.
```

  - q(s(s(X)), Y) :- q(X,Y). q(z,e). q(s(z),o).:

```
Callq = {q(X,Y) | p(X,Z) < Callp, Y < Term}
        + {q(X,Y) | q(s(s(X)),Y) < Callq}.
Retq  = {q(s(s(X)), Y) |
              q(A,Y) < Callq, A < UssX, q(X,Y) < Retq}
        + {q(z,e) | q(X,Y) < Callq, X < Uz, Y < Ue}
        + {q(s(z),o) | q(X,Y) < Callq, X < Usz, Y < Uo}.
```

  - r(e,yes). r(o,no).:

```
Callr = {r(Y,Z) | p(X,Z) < Callp, q(X,Y) < Retq}.
Retr  = {r(e,yes) | r(X,Y) < Callr, X < Ue, Y < Uyes}
        + {r(o,no)  | r(X,Y) < Callr, X < Uo, Y < Uno}.
```


Groundness Analysis [4]

- Ground and non ground terms:

```
Ground = {z} + {s(T) | T < Ground} + {o} + {e} + {yes} + {no}.

NotGround = {X | X < Var} + {s(T) | T < NotGround}.
```


- Groundness of r with unknown goals:

```
Goal = {p(X,Z) | X < Term, Z < Term}.
NotGroundr1 = {maybe | r(X,Y) < Callr, X < NotGround}.
NotGroundr2 = {maybe | r(X,Y) < Callr, Y < NotGround}.

Solution of variable NotGroundr1 :
----------------------------------
Empty Grammar

Solution of variable NotGroundr2 :
----------------------------------
S -> maybe
```

[4] without the "minor modifications to the set constraint algorithm" required in N. Heintze and J. Jaffar, *Set constraints and set-based analysis*, PPCP'94, LNCS 874, pp. 282–298, Springer, 1994.

- Groundness of `r` with ground goals:

```
Goal = {p(X,Z) | X < Ground, Z < Ground}.
NotGroundr1 = {maybe | r(X,Y) < Callr, X < NotGround}.
NotGroundr2 = {maybe | r(X,Y) < Callr, Y < NotGround}.

Solution of variable NotGroundr1 :
---------------------------------

Empty Grammar

Solution of variable NotGroundr2 :
---------------------------------

Empty Grammar
```


Closure Analysis [Pal95]

- Syntax : $e ::= 0 \mid \mathtt{succ}(e) \mid e_1 e_2 \mid \lambda x \bullet e$
- Basic constraints:

$\lambda x \bullet e$	$[\![\lambda x \bullet e]\!] \supseteq \{\lambda(x)\}$
0	$[\![0]\!] \supseteq \{\mathtt{int}\}$
$\mathtt{succ}(e)$	$[\![\mathtt{succ}\ e]\!] \supseteq \{\mathtt{int}\}$

- Connecting constraints:

$e_1 e_2$ for every $\lambda x \bullet e$ in e_0:

$$\Big(\lambda(x) \in [\![e_1]\!]\ ?\ [\![e_2]\!] \subseteq [\![x]\!] \wedge [\![e_1 e_2]\!] \supseteq [\![e]\!]\Big)$$

Reference

[Pal95] J. Palsberg. Closure analysis in constraint form. *TOPLAS*, 17(1):47–62, jan 1995.

- Palsberg & Schwartzbach example [PS95]:

$$\overbrace{\overbrace{(\lambda y \bullet \overbrace{(\overbrace{\ y\ }^{e_4}\ \overbrace{\ 0\ }^{e_5})}^{e_3})}^{e_1}\overbrace{(\lambda x \bullet \overbrace{\ x\ }^{e_6})}^{e_2}}^{e_0}$$

- By Tarski's fixpoint theorem, monotone constraints are equivalent to fixpoint equations.

Reference

[PS95] J. Palsberg and M.I. Schwartzbach. Safety analysis versus type inference. *Inf. & Comp.*, 118(1):128–141, 1995.

- Equations:

```
widen(Ce3,Cx,Cy).

% Closure analysis:

Ce0 =   {X | X < Cx,  lambda(x) < Ce1}
      + {X | X < Ce3, lambda(y) < Ce1}.
Ce1 = {lambda(y)}.
Ce2 = {lambda(x)}.
Ce3 =   {X | X < Cx,  lambda(x) < Cy}
      + {X | X < Ce3, lambda(y) < Cy}.
C0  = {int}.
Cx  =   {X | X < Ce2, lambda(x) < Ce1}
      + {X | X < C0,  lambda(x) < Cy}.
Cy  =   {X | X < Ce2, lambda(y) < Ce1}
      + {X | X < C0,  lambda(y) < Cy}.
```

Safety Analysis

- Closure analysis + safety constraints:

$$\begin{array}{ll} e_1 e_2 & [\![e_1]\!] \subseteq \{\lambda(x) \mid \lambda x \bullet e \in e_0\} \\ \mathtt{succ}(e) & [\![e]\!] \subseteq \{\mathtt{int}\} \end{array}$$

- Equations:

```
NotLambda = {int}.
NotInt = {lambda(x)} + {lambda(y)}.
Incorrect =   {X | X < Ce1, X < NotLambda}
            + {X | X < Cy,  X < NotLambda}.
```

- Solution:

```
Solution of variable Incorrect :
--------------------------------

Empty Grammar
```


Arithmetic Expression Interpreter

- Types:

```
Integer = {z} + {s(N) | N < Integer}.
Expr = {N | N < Integer} +
       {plus(E1, E2) | E1 < Expr, E2 < Expr}.
```

- Abstract syntax of the expression to evaluate:

```
ToEval = {plus(s(s(z)),plus(s(z),s(s(z))))}.
```

- Arithmetic expression interpreter:

```
Sum =   {call(E) | E < ToEval}
%% evaluation of constants
      + {return(N,N) |
                    call(N) < Sum, N < Integer}          .../...
```



```
%% evaluation of expression X+Y
%  evaluation of left sub-expression X
      + {call(X) | call(plus(X,Y)) < Sum}
%  evaluation of right sub-expression Y
      + {call(Y)| call(plus(X,Y)) < Sum,
                  return(X,N) < Sum, N < Integer}
%  evaluation of integer expression value(X)+ value(Y)
      + {call(plus(N,M)) |
                  call(plus(X,Y)) < Sum,
                  return(X,N) < Sum, N < Integer,
                  return(Y,M) < Sum, M < Integer}
% return computed value
      + {return(plus(X,Y),R) |
                  call(plus(X,Y)) < Sum,
                  return(X,N) < Sum, N < Integer,
                  return(Y,M) < Sum, N < Integer,
                  return(plus(N,M),R) < Sum}            .../...
```



```
%% addition of integers N+M
%  addition axiom: 0+N = N
      + {return(plus(z,N),N) |
                  call(plus(z,N)) < Sum, N < Integer}
% addition axiom: s(N)+M=s(N+M)
% 1) make recursive call:
      + {call(plus(N,M)) |
                  call(plus(s(N),M)) < Sum,
                  N < Integer, M < Integer}
% addition axiom: s(N)+M=s(N+M)
% 2) return value:
      + {return(plus(s(N),M),s(R)) |
                  call(plus(s(N),M)) < Sum,
                  N < Integer, M < Integer,
                  return(plus(N,M),R) < Sum}.
```

- Result:

```
Result = {R | E < ToEval, return(E,R) < Sum}.
```

- Using *dynamic partitionning*[5] [Bou92], the solution is:

```
Solution of variable Result :
----------------------------
- Grammar              - Constraints
     S -> s(S)               S.s = 5
     S -> z                  S.z = 1
```

References

[Bou92] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *J. Func. Prog.*, 2(4), 1992.

Lambda-calculus Interpreter

- Syntax:

```
Var = {f} + {g} + {h} +  {x} + {y} + {z}.
Exp =    {a} + {b} + {c(E1,E2) | E1 < Exp, E2 < Exp} % constants
       + {X | X < Var} + {lambda(F,E) | F < var, E < Exp}
       + {apply(E1,E2) | E1 < Exp, E2 < Exp}.
```

- Program analysis is obtained by encoding the eager lambda-calculus operational semantics in collecting form as a fix-point equation in the abtract domain;
- Approximation of computations follows from the fact that operations (like set union) are abstract. Various approximation levels in partitionning[6] and widening can be used;

[5] With A. Venet present implementation, dynamic partitionning had to be simulated "by hand".
[6] With A. Venet present implementation, partitionning had to be simulated "by hand".

- Example [Hei94]:

$$(\lambda f \bullet c((f\ a)(f\ b))\ \lambda x \bullet x)\ ;$$

- Abstract syntax:

```
ToEval = {apply(lambda(f,c(apply(f,a),apply(f,b))),lambda(x,x))}.
```

- Abstract interpreter:
  - Encoding[7]:

$$\langle [f:F, x:X], e, ?\rangle \longrightarrow \mathtt{call}(F, X, e)$$
$$\langle [f:F, x:X], e, v\rangle \longrightarrow \mathtt{return}(F, X, e, v)$$

Reference

[Hei94] N. Heintze. Set-based analysis of ML programs. LFP'94, 1994.


```
Eval = {call(error, error, E) | E < ToEval}
     + {call(F, X, E1) | call(F, X, apply(E1,E2)) < Eval}
     + {call(F, X, E2) | call(F, X, apply(E1,E2)) < Eval}
     + {return(F, X, lambda(Y,E), closure(F,X,lambda(Y,E))) |
          call(F, X, lambda(Y,E)) < Eval}
     + {call(F2, V2, E) |
          call(F1, X1, apply(E1,E2)) < Eval,
          return(F1, X1, E1, closure(F2,X2,lambda(x,E))) < Eval,
          return(F1, X1, E2, V2) < Eval}
     + {call(V2, X2, E) |
          call(F1, X1, apply(E1,E2)) < Eval,
          return(F1, X1, E1, closure(F2,X2,lambda(f,E))) < Eval,
          return(F1, X1, E2, V2) < Eval}
     + {return(F1, X1, apply(E1,E2), V) |
          call(F1, X1, apply(E1,E2)) < Eval,
          return(F1, X1, E1, closure(F2,X2,lambda(x,E))) < Eval,
          return(F1, X1, E2, V2) < Eval,
          return(F2, V2, E, V) < Eval}                 .../...
```

[7] General environments have to be encoded as lists. a preliminary abstract interpretation of the abstract equations can be used to get the above optimized simpler encoding as vectors.

```
    + {return(F1, X1, apply(E1,E2), V) |
         call(F1, X1, apply(E1,E2)) < Eval,
         return(F1, X1, E1, closure(F2,X2,lambda(f,E))) < Eval,
         return(F1, X1, E2, V2) < Eval,
         return(V2, X2, E, V) < Eval}
    + {return(F, X, x, X) | call (F, X, x) < Eval}
    + {return(F, X, f, F) | call (F, X, f) < Eval}
    + {call(F, X, E1) | call(F, X, c(E1, E2)) < Eval}
    + {call(F, X, E2) | call(F, X, c(E1, E2)) < Eval}
    + {return(F, X, c(E1,E2), c(V1,V2)) |
         return(F, X, E1, V1) < Eval,
         return(F, X, E2, V2) < Eval}
    + {return(F, X, a, a) | call(F, X, a) < Eval}
    + {return(F, X, b, b) | call(F, X, b) < Eval}.

Result = {V | return(error, error, E, V) < Eval, E < Expr}.
```


- Result:

```
Solution of variable Result :
-----------------------------
- Grammar
    S -> c(A,B)
    A -> a
    B -> b
- Constraints
    A.a = 1
    B.b = 1
    S.c = 1
```

Note: exactness is singular. In general recursion will lead to approximate results.

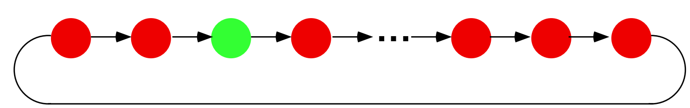
Type Checking

- The type checker is obtained by replacing basic values by their types (`int`, `bool`, ...) and keeping type closures[8];

```
...
  + {return(F, X, c(E1,E2), T) |                     % c: TxT->T
         return(F, X, E1, T) < Eval,
         return(F, X, E2, T) < Eval}
  + {return(F, X, a, int) | call(F, X, a) < Eval}    % b: int
  + {return(F, X, b, int) | call(F, X, b) < Eval}.   % a: int

Solution of variable Result :
-----------------------------
- Grammar          - Constraints
    S -> int           S.int = 1
```


Symbolic Model Checking [KMM+97][9]

- Variable-size composition of processes with synchronous communication along a ring;
- A circulating token is used to implement critical sections:



References

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Proc. 9th Int. Conf. CAV '97*. Springer-Verlag, 1997.

[8] Functional types are needed for type inference only.
[9] I thank Andreas Podelski for drawing my attention to this example.

Correctness Conditions

- [KMM+97] prove that at most one process resides in its critical section at any given instance (using an abstraction by regular expression);
- Do not prove that the token is not lost;
- Cannot prove that the number of processes is fixed (but unknown);
- For short, we make a further abstraction of the states considered by [KMM+97], by remembering which processes have (`y`) or do not have (`n`) a token.

1) Known number of processes

- Token moving right (1), y: process with token, n: process without token:

$$LynR \to LnyR, \qquad L, R \in (y \mid n)^\star$$

```
#fun trans = [           -> [           & [
               S -> y(S);   S -> y(S);    S.y - S'.y = 0; (1)
               S -> n(S);   S -> n(S);    S.n - S'.n = 0; (2)
               S -> nil;    S -> nil;     S.nil = 1;
              ]            ]              S'.nil = 1;
                                         ]
```

(1) Number of y in $LynR$ = Number of y in $LnyR$;

(2) Number of n in $LynR$ = Number of n in $LnyR$.

- Token moving right (2):

$$nMy \to yMn, \qquad M \in (y \mid n)^\star$$

```
#fun transl = [           -> [           & [
                S -> n(A);   S -> y(A);    S.n = 1;
                A -> y(A);   A -> y(A);    A.nil = 1;
                A -> n(A);   A -> n(A);    S'.y = 1;
                A -> nil;    A -> nil;     A'.nil = 1;
               ]            ]              A.y - A'.y = 1;  (1)
                                           A.n - A'.n = -1; (2)
                                          ]
```

(1) Number of y in Mn = Number of y in $M'n + 1$;

(2) Number of n in Mn = Number of n in $M'n - 1$.

- Initial states:

$$yn^{99}$$

```
#def Init = [           & [
             S -> y(A);    S.y = 1;
             A -> n(A);    A.n = 99;
             A -> nil;   ]
            ]
```

- Reachable states:

```
widen(Reach).
Reach = Init + trans(Reach) + transl(Reach) .
```

```
Solution of variable Reach :
----------------------------
- Grammar
    S -> n(S)
    S -> y(S)
    S -> nil
- Constraints
    S.n = 99
    S.y = 1
```

The Futurebus+ is an example of network process of regularly connected finite-state processes which was verified for many configurations with bugs later discovered by analyzing additional larger configurations. It is necessary to consider *uniform verification, for all possible configurations*.

2) Unknown number of processes

- Initial state, yn^i, i unknown:

```
#def Init = [          & [
   S -> state(I,A);      S.state = 1;
   I -> z;               I.z = 1;
   I -> s(I);            A.y = 1;          (1)
   A -> y(B);            I.s - B.n = -1;   (2)
   B -> n(B);          ]
   B -> nil;
]
```

- `I` $=$ `s(s(s(...(z)...)))` is a counter encoding the initial number of processes;

(1) There is initially one process with a token (`y`);

(2) There is initially `I` -1 processes without token (`n`);

- $I \times LynR \quad \rightarrow \quad I \times LnyR, \qquad L, R \in (y \mid n)^\star$

```
#fun trans =
[                   -> [                & [
 S -> state(I,A);     S -> state(I,A);    S.state = 1;,I.z = 1;
 I -> z;              I -> z;             S'.state = 1;
 I -> s(I);           I -> s(I);          I'.z = 1;
 A -> y(A);           A -> y(A);          A.y - A'.y =  0; (1)
 A -> n(A);           A -> n(A);          A.n - A'.n = 0;  (2)
 A -> nil;            A -> nil;           I.s - I'.s = 0;  (3)
]                   ]                     A.nil = 1;
                                          A'.nil = 1;
                                        ]
```

(1) Number of y in $LynR$ = Number of y in $LnyR$;

(2) Number of n in $LynR$ = Number of n in $LnyR$;

(3) The <u>initial</u> number I of processes is constant.

- $I \times nMy \quad \rightarrow \quad I \times yMn, \qquad M \in (y \mid n)^\star$

```
#fun transl =
[                   -> [                & [
 S -> state(I,A);     S -> state(I,A);    S.state = 1; I.z = 1;
 I -> z;              I -> z;             S'.state = 1; I'.z = 1;
 I -> s(I);           I -> s(I);          B.nil = 1; B'.nil = 1;
 A -> n(B);           A -> y(B);          A.n = 1;
 B -> y(B);           B -> y(B);          A'.y = 1;
 B -> n(B);           B -> n(B);          B.y - B'.y = 1;  (1)
 B -> nil;            B -> nil;           B.n - B'.n = -1; (2)
]                   ]                     I.s - I'.s = 0;  (3)
                                        ]
```

(1) Number of y in Mn = Number of y in $M'n + 1$;

(2) Number of n in Mn = Number of n in $M'n - 1$.

(3) The <u>initial</u> number I of processes is constant.

- Reachable states:

```
widen(Reach).
Reach = Init + trans(Reach) + transl(Reach) .

Solution of variable Reach :
----------------------------
- Grammar              - Constraints
     S -> state(A,B)      S.state = 1
     A -> z               A.z = 1
     A -> s(A)            B.y = 1      (1)
     B -> n(B)            B.n-A.s = 1  (2)
     B -> y(B)
     B -> nil
```

(1) One process, marqued `y`, has the token;

(2) There are $n-1$ processes, marqued `n`, without token, where n is the initial number of processes.

Towards Universal Abstract Interpreters

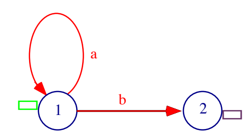
(Generic) Abstract Interpreter

- The classical abstract interpreter:

Program —**Equation generator**→ System of equations —**Solver**→ Solution

- The system of equations is an abstraction of the concrete semantics;
- With generic abstract interpreters, the solver has an additional abstract domain parameter (replaced by a universal abstract domain).

Example: traces of a transition system



- A small-step operational semantics is specified as a set of transitions:

$$
\begin{aligned}
s, s' &\in \texttt{States} \\
a &\in \texttt{Actions} \\
t &\in \texttt{Transitions} \\
t &::= \texttt{trans}(s, a, s')
\end{aligned}
$$

- A trace semantics is specified as a set of traces. Traces are finite sequences of successive transitions:

$$T, T' \in \texttt{Traces}$$
$$T ::= \texttt{emptyt} \mid \texttt{trace}(T', t)$$

For example, the trace:

$$(1, \mathtt{a}, 2)(2, \mathtt{b}, 3)(3, \mathtt{c}, 4)$$

is encoded as:

$$\texttt{trace}(\texttt{trace}(\texttt{trace}(\texttt{emptyt}, \texttt{trans}(1, \mathtt{a}, 2)), \texttt{trans}(2, \mathtt{b}, 3)), \texttt{trans}(3, \mathtt{c}, 4))$$

- `PrefixTraces` is the prefix-closed set of execution traces of the given automaton for the given initial state `1`:

```
PrefixTraces = {trace(emptyt,trans(one,a,one))}
             + {trace(emptyt,trans(one,b,two))}
             + {trace(trace(T,trans(S,A,one)),trans(one,a,one)) |
                    trace(T,trans(S,A,one)) < PrefixTraces}
             + {trace(trace(T,trans(S,A,one)),trans(one,b,two)) |
                    trace (T,trans(S,A,one)) < PrefixTraces}.
```

- `CompleteTraces` is the set of traces complete traces terminating in the given final state `2`:

```
CompleteTraces = {trace(T,trans(S,A,two)) |
                    trace(T,trans(S,A,two)) < PrefixTraces}.
```

- The abstraction is provided by the widening: `widen(PrefixTraces)`.

- Using *static partitionning* [Cou81] along the actions [10], the solver will produce the solution $\epsilon(1, a, 1)^\star(1, b, 2)$:

```
Solution of variable CompleteTraces :
-------------------------------------
- Grammar                                  - Constraints
     S -> trace(A,B)     G -> one               A.emptyt = 1
     A -> trace(A,C)     H -> a                 H.a-I.one = 0
     A -> emptyt         I -> one               E.b = 1
     B -> trans(D,E,F)                          B.trans = 1
     C -> trans(G,H,I)                          C.trans-I.one = 0
     D -> one                                   A.trace-I.one = 0
     E -> b                                     F.two = 1
     F -> two                                   S.trace = 1
```

References

[Cou81] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, 1981.

Universal Abstract Interpreter (I)

Program —**Equation generator**→ System of equations for concrete semantics + abstraction function —**&**→ System of equations for abstract semantics —**Solver**→ Solution

[10] With A. Venet present implementation, static partitionning had to be simulated "by hand".

Example: execution traces semantics analysis parameterized by a small-step operational semantics

- The prefix-closed set `PrefixTraces` of execution traces is parameterized by a set of initial states `InitStates` and a small-step operational semantics `Semantics`:

```
PrefixTraces = {trace(emptyt,trans(S1,A,S2)) | S1 < InitStates,
                        trans(S1,A,S2) < Semantics}
             + {trace(trace(T,trans(S1,A,S2)),trans(S2,B,S3)) |
                        trace(T,trans(S1,A,S2)) < PrefixTraces,
                        trans(S2,B,S3) < Semantics}.
```

- The set of complete traces is parameterized by final states:

```
CompleteTraces = {trace(T,trans(S1,A,S2)) |
                        trace(T,trans(S1,A,S2)) < PrefixTraces,
                        S2 < FinalStates}.
```


- A specific analysis is obtained by providing the `Semantics`, `InitStates` and `FinalStates` actual parameters:

```
Semantics   = {trans(one,a,one)} + {trans(one,b,two)}.
InitStates  = {one}.
FinalStates = {two}.
```


Universal Abstract Interpreter (II)

Program —**Equation generator**→ Abstract syntax —**&**→

System of equations concrete semantics & abstraction function —**&**→ System of equations abstract semantics

—**Solver**→ Solution

Example: execution traces semantics analysis parameterized by an abstract syntax

- For a specific analysis, the equation generator produces the abstract syntax of the automaton:

```
InitStateList  = {statelist(one,emptys)}.
FinalStateList = {statelist(two,emptys)}.
TransitionList = {translist(trans(one,a,one),
                   translist(trans(one,b,two), emptyl))}.
Automaton = {automaton(I,F,T) |  I < InitStateList,
                 F < FinalStateList, T < TransitionList}.
```

- Abstract syntax to concrete semantics mapping:

```
ExtractTrans =   {transition(T,L) |
                    automaton(I,F,translist(T,L)) < Automaton}
               + {transition(U,L) |
                    transition(R,translist(U,L)) < ExtractTrans}.

ExtractInit =   {initstate(S,L) |
                   automaton(statelist(S,L),F,T) < Automaton}
              + {initstate(S,L) |
                   initstate(R,initstate(S,L)) < ExtractInit}.

ExtractFinal =   {finalstates(S,L) |
                    automaton(I,statelist(S,L),T) < Automaton}
               + {finalstates(S,L) |
                    finalstates(R,statelist(S,L)) < ExtractFinal}.
```


- The small-step operational semantics of the automaton is mapped from the abstract syntax:

```
InitStates  = {S | initstate(S,L) < ExtractInit}.
FinalStates = {S | finalstates(S,L) < ExtractFinal}.
Semantics   = {T | transition(T,L) < ExtractTrans}.
```

- Complete traces abstract semantics (unchanged):

```
PrefixTraces = {trace(emptyt,trans(S1,A,S2)) | S1 < InitStates,
                  trans(S1,A,S2) < Semantics}
             + {trace(trace(T,trans(S1,A,S2)),trans(S2,B,S3)) |
                  trace(T,trans(S1,A,S2)) < PrefixTraces,
                  trans(S2,B,S3) < Semantics}.
CompleteTraces = {trace(T,trans(S1,A,S2)) |
                    trace(T,trans(S1,A,S2)) < PrefixTraces,
                    S2 < FinalStates}.
```


- The equation generator is reduced to a minimum;
- If the abstract domain is expressive enough, one can express semantics at any level of abstraction (e.g. from operational to axiomatic semantics);
- By using intermediate languages [11], one can write portable meta-interpreters;
- The design of a general-purpose language for expressing and approximating semantics remains to be done [12, 13, 14];
- The same intermediate language could be interpreted in a hierarchy of abstract domains to adjust the cost/benefit ratio.

Efficiency

[11] Automata in our simplistic exemple.
[12] This would be similar to the emergence of intermediate languages as commonly used in compiler
[13] The abstract language considered in this talk is only a very first step.
[14] The λ-calculus with denotational semantics is not flexible enough in my opinion.

Anticipated Difficulties

- It is difficult to imagine that spectacular optimisations can lead to universal analysers as efficient as special purpose analysers;
- The ideas presented here might remain useful only for rapid prototyping;
- High ambitions hopefully stimulate research!

A few hints toward efficiency

- Compiler optimisation techniques are directly applicable to optimise the abstract interpreter specification:
  - Abstract interpretation (constant propagation, neediness, etc...),
  - Partial evaluation (static partionning, ...) & program transformation;
- Certainly requires more research on involved algorithmic aspects of abstract interpretation;
- Might be easier than automatic compiler generalization.

Conclusion

- Can we conceive an abstract interpretation based model of approximated computation?
- Can we design a meta language to express a hierarchy of abstract semantics of programming languages which implementations, at different level of abstraction, would lead to practical abstract interpreters?
- Something between a stimulating research challenge and a dream!

Contents