Computational Logic

An Introduction to Abstract Interpretation

(and Abstract Interpretation of Logic Programs)

Introduction

- Many CS problems related to program analysis / synthesis
- Alternatively: derive properties which hold for program P (program analysis)
- Prove that some property holds for program P (program analysis for verification)
- Given a program P, generate a program P' which is
  - ◇ in some way equivalent to P
  - ◇ behaves better than P w.r.t. some criteria

  Typical approach:
  - ◇ *identify* that some *invariant* holds, and
  - ◇ *specialize* the program for the particular case

  (program analysis for program transformation and synthesis)

Program Analysis

- Frequent in compilers although seldom treated in a formal way:
  - ◇ “code optimization”,
  - ◇ “dead code elimination”,
  - ◇ “code motion”,
  - ◇ ...

  [Aho, Ullman 77]
- Often referred to as “dataflow analysis”
- Abstract interpretation provides a formal framework for developing program analysis tools
- Analysis phase + synthesis phase $\equiv$ Abstract Interpretation + (abstract) Program Transformation

What is abstract interpretation?

- Consider detecting that one branch will not be taken in:
  $\textbf{int}\ x, y, z;\quad y := read(file);\quad x := y * y;$
  if $x\ \geq\ 0$ **then** $z := 1$ **else** $z := 0$
  - ◇ Exhaustive analysis in the standard domain: non-termination
  - ◇ Human reasoning about programs – uses abstractions or approximations: signs, order of magnitude, odd/even, ...
  - ◇ Basic Idea: use *approximate* (generally *finite*) representations of computational objects to make the problem of program dataflow analysis *tractable*
- Abstract interpretation is a formalization of this idea:
  - ◇ define a non-standard semantics which can approximate the *meaning* or *behaviour* of the program in a finite way
  - ◇ expressions are computed over an approximate (abstract) domain rather than the concrete domain (i.e., meaning of operators has to be reconsidered w.r.t. this new domain)

Comparison to other methods

- Very general:
  can be applied to any language with well defined (procedural or declarative) semantics
- Automatic – (vs. proof methods)
- Static – not all possible runs actually tried (vs. model checking)
- Sound – no possible run omitted (vs. debugging)

Example: integer sign arithmetic

- Consider the domain $D = Z$ (integers)
  and the multiplication operator: $* : Z^2 \rightarrow Z$
- We define an "abstract domain:" $D_\alpha = \{[-], [+]\}$

  and abstract multiplication: $*_\alpha : D_\alpha^2 \rightarrow D_\alpha$ defined by:

$*_\alpha$	$[-]$	$[+]$
$[-]$	$[+]$	$[-]$
$[+]$	$[-]$	$[+]$

- This allows us to conclude, for example, that $y = x^2 = x * x$ is never negative
- Some observations:
  - ◇ The basis is that whenever we have $z = x * y$ then:
    if $x, y \in Z$ are approximated by $x_\alpha, y_\alpha \in D_\alpha$
    then $z \in Z$ is approximated by $z_\alpha = x_\alpha *_\alpha y_\alpha$
  - ◇ It is important to formalize this notion of approximation,
    in order to be able to prove an analysis correct
  - ◇ Approximate computation is generally less precise but faster (tradeoff)
  - ◇ In interesting cases such "speed differential" is often extreme: i.e., termination vs. non-termination

Example: integer sign arithmetic (Contd.)

- Again, $D = Z$ (integers)
  and: $* : Z^2 \to Z$
- Let's define a *more refined* "abstract domain": $D'_\alpha = \{[-], [0], [+]\}$
- Abstract multiplication: $*_\alpha : D'^2_\alpha \to D'_\alpha$ defined by

$*_\alpha$	$[-]$	$[0]$	$[+]$
$[-]$	$[+]$	$[0]$	$[-]$
$[0]$	$[0]$	$[0]$	$[0]$
$[+]$	$[-]$	$[0]$	$[+]$

- This now allows us to reason that $z = y * (0 * x)$ is zero
- Some observations:
  - ◇ There is a degree of freedom in defining different abstract operators and domains
  - ◇ The minimal requirement is that they be "safe" or "correct"
  - ◇ Different "safe" definitions result in different kinds of analyses

Example: integer sign arithmetic (Contd.)

- Again $D = Z$ (integers)
  and the *addition* operator: $+ : Z^2 \rightarrow Z$
- We cannot use $D'_\alpha = \{[-], [0], [+]\}$ because we wouldn't know how to represent the result of $[+] +_\alpha [-]$
  (i.e., our abstract addition would not be closed)
- New element "$\top$" (supremum): approximation of any integer
- New "abstract domain": $D''_\alpha = \{[-], [0], [+], \top\}$
- Abstract addition: $+_\alpha : D''^2_\alpha \rightarrow D''_\alpha$ defined by:

$+_\alpha$	$[-]$	$[0]$	$[+]$	$\top$
$[-]$	$[-]$	$[-]$	$\top$	$\top$
$[0]$	$[-]$	$[0]$	$[+]$	$\top$
$[+]$	$\top$	$[+]$	$[+]$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

... (alt:

$+_\alpha$	$[-]$	$[0]$	$[+]$	$\top$
$[-]$	$\top$	$\top$	$\top$	$\top$
$[0]$	$\top$	$\top$	$\top$	$\top$
$[+]$	$\top$	$\top$	$\top$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

)

- We can now reason that $z = x^2 + y^2$ is never negative

Important observations

- In addition to the imprecision due to the coarseness of D_α, the abstract versions of the operations (dependent on D_α) may introduce further imprecision
- Thus, the choice of *abstract domain* and the definition of the *abstract operators* are crucial

Issues in Abstract Interpretation

- Required:
  - ◇ Correctness – safe approximations: because most “interesting” properties are undecidable the analysis necessarily has to be approximate. We want to ensure that the analysis is “conservative” and errs on the “safe side”
  - ◇ Termination – compilation should definitely terminate

  (Note: not always the case in everyday program analysis tools!)

- Desirable – “practicality”:
  - ◇ Efficiency – in practice finite analysis time is not enough: finite *and* small
  - ◇ Accuracy – of the collected information: depends on the appropriateness of the abstract domain and the level of detail to which the interpretation procedure mimics the semantics of the language
  - ◇ “Usefulness” – determines which information is worth collecting

Safe Approximations

- Basic idea in approximation: for some property p we want to show that

$$\forall x, x \in S \Rightarrow p(x)$$

  Alternative: construct a set $S_a \supseteq S$, and prove

$$\forall x, x \in S_a \Rightarrow p(x)$$

  then, S_a is a *safe approximation* of S

- Approximation on functions: for some property p we want to show that

$$\forall x, x \in S \Rightarrow p(F(x))$$

- A function

$$G : S \rightarrow S$$

  is a *safe approximation* of F if

$$\forall x, x \in S, p(G(x)) \Rightarrow p(F(x))$$

Approximation of the meaning of a program

- Let the meaning of a program P be a mapping F_P from input to output, input and output values $\in$ “standard” domain D:

$$F_P : D \rightarrow D$$

- Let’s ‘lift’ this meaning to map sets of inputs to sets of outputs

$$F_P^* : \wp(D) \rightarrow \wp(D)$$

  where $\wp(S)$ denotes the powerset of S, and

$$F_P^*(S) = \{F_P(x) | x \in S\}$$

- A function

$$G : \wp(D) \rightarrow \wp(D)$$

  is a *safe approximation* of F_P^* if

$$\forall S, S \in \wp(D), G(S) \supseteq F_P^*(S)$$

- Properties can be proved using G instead of F_P^*

Approximation of the meaning of a program (Contd.)

- For some property p we want to show that

  for some inputs S, $p(F_P^*(S))$

- We show that

  for some inputs S_a, $p(G(S_a))$

- Since $G(S_a) \supseteq F_P^*(S_a)$

  for some inputs S_a, $p(F_P^*(S_a))$

  (Note: abuse of notation – F_P^* does not work on abstract values S_a)

- As long as F_P^* is monotonic:

$$S_a \supseteq S \Rightarrow F_P^*(S_a) \supseteq F_P^*(S)$$

- And since $S_a \supseteq S$, then:

  for some inputs S, $p(F_P^*(S))$

Abstract Domain and Concretization Function

- The domain $\wp(D)$ can be represented by an “abstract” domain D_α of finite representations of (possibly) infinite objects in $\wp(D)$
- The representation of $\wp(D)$ by D_α is expressed by a (monotonic) function called a *concretization function*:

$$\gamma : D_\alpha \rightarrow \wp(D)$$

such that $\gamma(\lambda) = d$ if d is the largest element (under $\subseteq$) of $\wp(D)$ that λ describes
[$(\wp(D), \subseteq)$ is obviously a complete lattice]

E.g., in the “signs” example, with $D_\alpha = \{[-], [0], [+], \top\}$, γ is given by

$$\begin{array}{ll} \gamma([-]) & = \{x \in Z \mid x < 0\} \\ \gamma([0]) & = \{0\} \\ \gamma([+]) & = \{x \in Z \mid x > 0\} \\ \gamma(\top) & = Z \end{array}$$

- $\gamma(?) = \emptyset \rightarrow$ we define $\bot \mid \gamma(\bot) = \emptyset$

Abstraction Function

- We can also define (not strictly needed) a (monotonic) *abstraction function*

$$\alpha : \wp(D) \to D_\alpha$$

$\alpha(d) = \lambda$ if λ is the “least” element of D_α that describes d
[under a suitable ordering defined on the elements of D_α]

e.g., in the “signs” example,

$$\begin{aligned}
\alpha(\{1,2,3\}) &= [+] \text{ (and not } \top\text{)} \\
\alpha(\{-1,-2,-3\}) &= [-] \text{ (and not } \top\text{)} \\
\alpha(\{0\}) &= [0] \\
\alpha(\{-1,0,1\}) &= \top
\end{aligned}$$

$$\begin{tikzcd}
\wp(D) \arrow[r, shift left, "\alpha"] & D_\alpha \arrow[l, shift left, "\gamma"]
\end{tikzcd}$$

Abstract Meaning and Safety

- We can now define an abstract meaning function as

$$F_\alpha : D_\alpha \to D_\alpha$$

which is then safe if

$$\forall \lambda, \lambda \in D_\alpha, \gamma(F_\alpha(\lambda)) \supseteq F_P^*(\gamma(\lambda))$$

$$\begin{tikzcd}
\lambda \arrow[rr, "\gamma"] \arrow[d, "F_\alpha"'] & & d \arrow[ll, bend left, "\alpha"] \arrow[d, "F_P^*"] \\
\lambda_r \arrow[r, "\gamma"'] & \circ \arrow[r, phantom, "\supseteq"] & r
\end{tikzcd}$$

- We can then prove a property of the output of a given class of inputs represented by λ by proving that all elements of $\gamma(F_\alpha(\lambda))$ have such property
- E.g., in our example, a property such as “if this program takes a positive number it will produce a negative number as output” can be proved

Proving properties in the abstract

- Generating F_α:
  - ◇ F_P obtained from program and predefined semantics of operators
    $(x + z) * 3$, $F_P = (x + z) * 3$
  - ◇ Automatic analysis:
    F_α should be obtainable from program and semantics of abstract operators (compositional properties)
    $\{odd, even, +_\alpha, *_\alpha\} \Rightarrow F_\alpha = (x +_\alpha z) *_\alpha odd$
- “If this program takes a positive number it will produce a negative number as output”
  - ◇ $P = (y := x * -3)$, x input, y output
  - ◇ $F_P = x * -3$
  - ◇ $F_\alpha = x *_\alpha [-]$
  - ◇ $F_\alpha([+]) = [+] *_\alpha [-] = [-]$

Collecting Semantics

- "Input-output" semantics often too coarse for useful analysis: information about "state" at *program points* generally required $\rightarrow$ "extended semantics"
- Program points can be reached many times, from different points, and in different "states" $\rightarrow$ "collecting" ("sticky") semantics

$$\{x > 3\}\ y := x * -3\ \{y < -9\} \text{ or } \{x < -3\}\ y := x * -3\ \{y > 9\}$$

$$\{x = [+]\}\ y := x * -3\ \{y = [-]\} \text{ or } \{x = [-]\}\ y := x * -3\ \{y = [+]\}$$

- Analysis often computes a collection of abstract states for a program point

$$\{x = \{[+], [-]\}\}\ y := x * -3\ \{y = \{[-], [+]\}\}$$

- Often more efficient to "summarize" states into one which gives the best overall description $\rightarrow$ lattice structure in abstract domain

$$\{x = \sqcup\{[+], [-]\}\}\ y := x * -3\ \{y = \sqcup\{[-], [+]\}\}$$

Lattice Structure

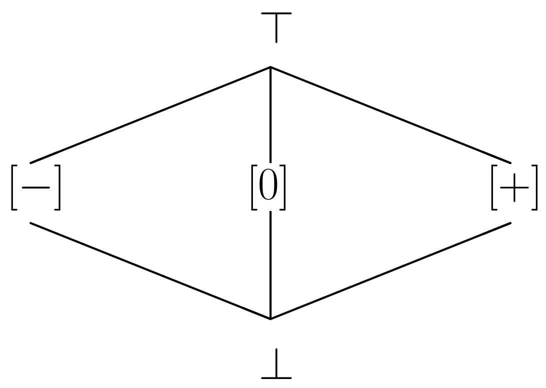
- The ordering on $\wp(D)$, $\subseteq$, induces an ordering on D_α, $\leq_\alpha$ ("approximates better")
  E.g., we can choose either $\alpha(\{1,2,3\}) = [+]$ or $\alpha(\{1,2,3\}) = \top$,
  but $\gamma([+]) = \{x \in Z | x > 0\}$ and $\gamma(\top) = Z$, and
  since $\{x \in Z | x > 0\} \subseteq Z$ we have $[+] \leq_\alpha \top$, i.e., $[+]$ approximates better than $\top$,
  it is more precise
- It is generally required that $(D_\alpha, \leq_\alpha)$ be a complete lattice
- Therefore, for all $S \subseteq D_\alpha$ there exists a unique least upper bound $\sqcup S \in D_\alpha$ –i.e., such that
  - ◇ $\forall \lambda_s \in S, \lambda_s \leq_\alpha \sqcup S$
  - ◇ $(\forall \lambda_s \in S, \lambda_s \leq_\alpha \lambda) \Rightarrow \sqcup S \leq_\alpha \lambda$
- Intuition: given a set of approximations of the "current state" at a given point in a program, to ensure that it is the best "overall" description for the point:
  - ◇ $\sqcup S$ approximates *everything* the elements of S approximate
  - ◇ $\sqcup S$ is the best approximation in D_α

Example: integer sign arithmetic

- We consider $D_\alpha = \{[-], [0], [+], \top\}$
  - ◇ We add $\perp$ (infimum) so that $\alpha(\emptyset)$ exists and to have a complete lattice: $D_\alpha = \{\perp, [-], [0], [+], \top\}$
  - ◇ (Intuition:
    it represents a program point that is never reached)
  - ◇ The concretization function has to be extended with

$$\gamma(\perp) = \emptyset$$

  - ◇ The lattice is then given by:



  - ◇ $\sqcup\{[+], [-]\} = \sqcup\{[-], [+]\} = \top$

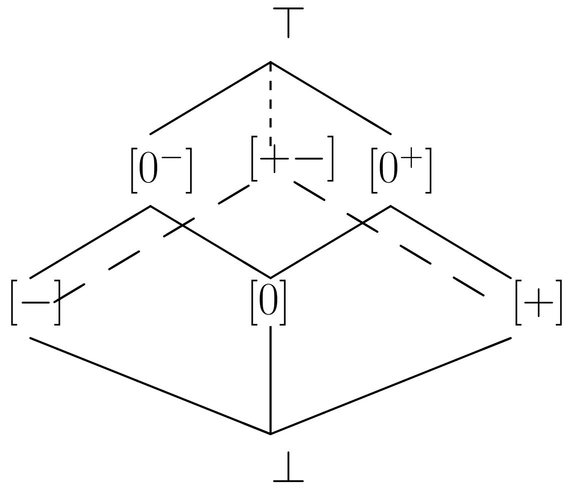
Example: integer sign arithmetic (Contd.)

- To make $\sqcup$ more meaningful we consider $D_\alpha = \{\perp, [-], [0^-], [0], [0^+], [+], \top\}$

$$\begin{array}{llllll}
\gamma(\perp) & = \emptyset & \gamma(\top) & = Z & & \\
\gamma([-]) & = \{x \in Z | x < 0\} & \gamma([+]) & = \{x \in Z | x > 0\} & \gamma([0]) & = \{0\} \\
\gamma([0^-]) & = \{x \in Z | x \leq 0\} & \gamma([0^+]) & = \{x \in Z | x \geq 0\} & &
\end{array}$$

- The lattice is then given by:

$\sqcup\{[+], [-]\} = \top$?



- $\sqcup\{[-], [0]\} = [0^-]$
  accurately represents a program point where a variable can be negative or zero

The Galois Insertion Approach

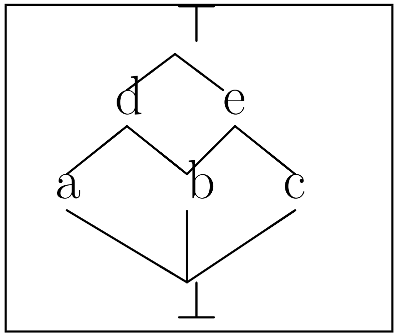
- In the following, we will refer to $\wp(D)$ simply as D
- (Collecting) program semantics is often given as $\mathit{lfp}\ (F)$ (the least S s.t. $S = F(S)$, F being the program-dependent semantic function on D)
- Thus, we need to relate this fixpoint to (that of) the approximate semantic function F_α (which approximates F and operates on elements of an abstract domain D_α)
- Assume: D and D_α are complete lattices; $\gamma : D_\alpha \rightarrow D$ and $\alpha : D \rightarrow D_\alpha$ are monotonic functions. The structure $(D_\alpha, \gamma, D, \alpha)$ is called a *Galois Insertion* if:
  - ◇ $\forall \lambda \in D_\alpha . \lambda = \alpha(\gamma(\lambda))$
  - ◇ $\forall d \in D . d \subseteq \gamma(\alpha(d))$
- *Safe approximation*, defined now in terms of a Galois insertion:
  Let a Galois insertion $(D_\alpha, \gamma, D, \alpha)$, $\lambda \in D_\alpha$ *safely approximates* $d \in D$ iff $d \subseteq \gamma(\lambda)$
- Fundamental Theorem [Cousot]: Given a Galois insertion $(D_\alpha, \gamma, D, \alpha)$, and two (monotonic) functions $F : D \rightarrow D$ and $F_\alpha : D_\alpha \rightarrow D_\alpha$ then if F_α approximates F, $\mathit{lfp}\ (F_\alpha)$ approximates $\mathit{lfp}\ (F)$

Termination: conditions on F_α and D_α

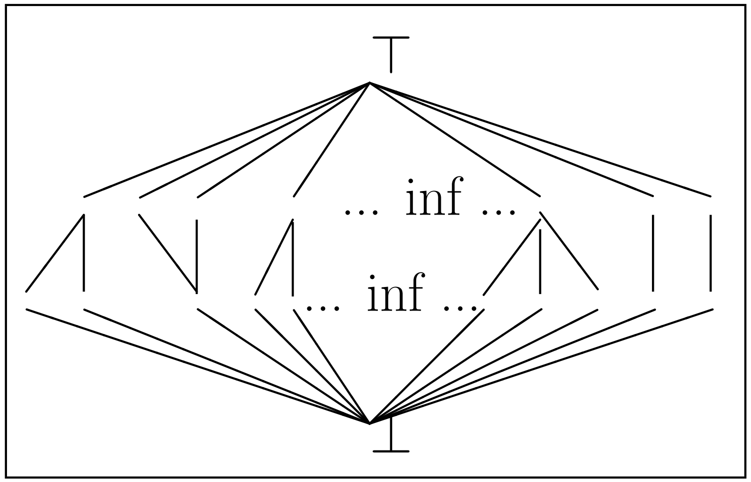
- The question is whether $\mathit{lfp}\ (F_\alpha)$ is finitely computable
- The abstract operator F_α operates on elements of an abstract domain D_α, which we have required to be a complete lattice,
  and F_α is monotonic, therefore
  $\mathit{lfp}\ F_\alpha = F_\alpha \uparrow n$
  for some n which we would like to be finite
  (i.e., we would like the Kleene sequence to be finite)
- Recalling the characteristics of fixpoints on lattices, the Kleene sequence will be finite in cases including:
  - ◇ D_α is finite
  - ◇ D_α is ascending chain finite

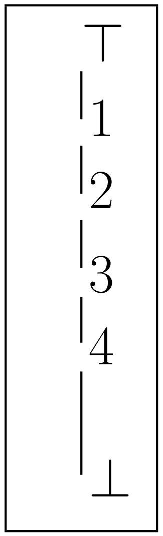
Lattice Structures

finite



finite_depth





ascending chain finite

Termination: Discussion

- Showing monotonicity of F_α may be more difficult than showing that D_α meets the finiteness conditions
- There may be an F_α which terminates even if the conditions are not met
- Conditions may also be relaxed by restricting the class of programs (e.g., non-recursive/non-looping programs pose few difficulties, although they are hardly interesting)
- In some cases an approximation from above ($\mathit{gfp}\ (F_\alpha)$) can also be interesting
- There are other alternatives to finiteness: dynamic bounded depth, etc. (See: Widening and Narrowing)

Origins (General Programming)

- The idea itself (i.e., rule of signs) predates computation...
- The idea of computing by approximations was used as early as 1963 by Naur ("pseudo evaluation", in the Gier Algol compiler),

  "a process which combines the operators and operands of the source text in the manner in which an actual evaluation would have to do it, but which operates on descriptions of the operands, not on their values"
- 1972, Sintzoff (proving well-formedness and termination properties)
- 1975, Wegbreit appears to be the first to develop a lattice-theoretic model
- Mid 70's: Kam, Kindall, Tarjan, Ullman, ...
- 1976,77, Patrick and Radhia Cousot proposed a formal model for the analysis of imperative ("flowchart") languages: unifying framework
  - ◇ Define a "static" semantics: associate a set of possible storage states with each program point
  - ◇ Dataflow analysis constructed then as a finitely computable approximation to the static semantics

Analyzing Logic Programs

- Why logic programs?
  - ◇ Because it is a very cool programming paradigm
  - ◇ Because if you can analyze full Prolog well you know how to analyze any language
  - ◇ Because if you have an analyzer for full Prolog you can analyze any language with it (cf., “transformation to Horn clauses”)

    (This idea was the basis of the CiaoPP analyzer and is quite popular today)

Analyzing Logic Programs

- Which semantics?
  - ◇ Declarative semantics: concerned with what is a consequence of the program
    - * Model-theoretic semantics
    - * Fixpoint (T_P operator-based) semantics

    can in some cases be what the program actually does
    (cf. database-style bottom-up evaluation)
  - ◇ Operational semantics: close to the behavior of the program
    - * SLD-resolution based (success sets)
    - * Denotational
    - * Can cover possibilities other that SLD: reactive, parallel, ...
- Analyses based on declarative semantics are often called "bottom up" analyses
- Analysis based on the (top-down) operational semantics are often called "top-down" analyses
- Also, intermediate cases (generally achieved through program transformation)

Case Study: Fixpoint Semantics

- Given the first-order language L associated with a given program P, the *Herbrand universe* (U) is the set of all *ground terms* of L.
- The *Herbrand Base* (B) is the set of all *ground atoms* of L.
- A *Herbrand Interpretation* is a subset of B.
  I is the set of all Herbrand interpretations ($\wp(B)$).
- A *Herbrand Model* is a Herbrand interpretation which contains all logical consequences of the program.
- The *Immediate Consequence Operator* (T_P) is a mapping $T_P : I \rightarrow I$ defined by:

  $$T_P(M) = \{h \in B \mid \exists C \in ground(P),\ \ C = h \leftarrow b_1, ..., b_n \ \ \text{and } b_1, \ldots b_n \in M\}$$

  (in particular, if $(a \leftarrow) \in P$, then $ground(a) \subseteq T_P(M)$, for every M).
- T_P is monotonic, so it has a least fixpoint $lfp(T_P)$ which can be obtained as $T_P{\uparrow}\omega$ starting from the bottom element of the lattice (the empty interpretation, $\emptyset$).
- (Characterization Theorem) [Van Emden and Kowalski]:
  The Least Herbrand Model of P, H is $\mathit{lfp}\ (T_P)$

Fixpoint Semantics: Example

- Example:

$$
\begin{aligned}
P = \{\ & p(f(X)) \leftarrow p(X). \\
& p(a). \\
& q(a). \\
& q(b).\ \}
\end{aligned}
$$

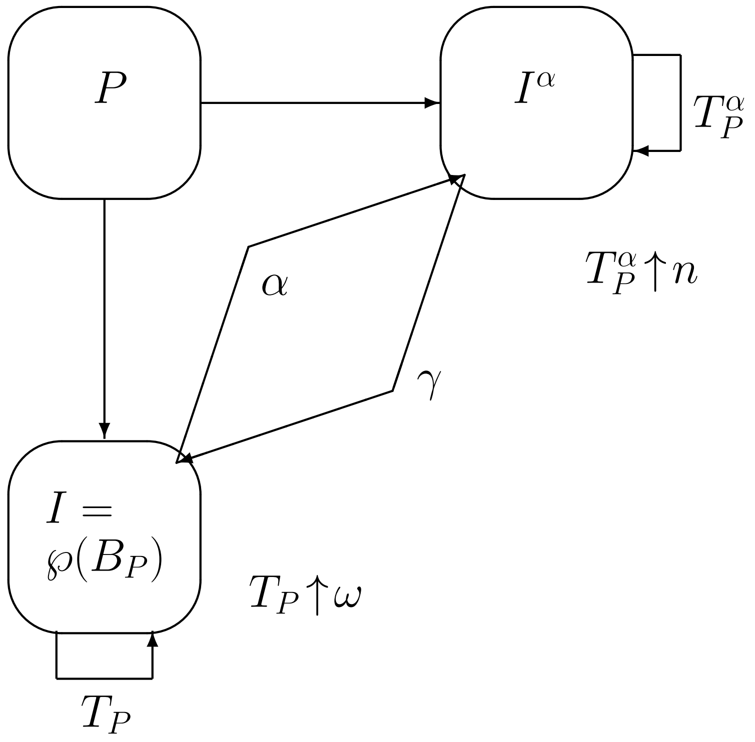
$$
\begin{aligned}
U &= \{a, b, f(a), f(b), f(f(a)), f(f(b)), \ldots\} \\
B &= \{p(a), p(b), q(a), q(b), p(f(a)), p(f(b)), q(f(a)), \ldots\} \\
I &= \textit{all subsets of } B \\
H &= \{q(a), q(b), p(a), p(f(a)), p(f(f(a))), \ldots\}
\end{aligned}
$$

$$
\begin{aligned}
&T_P{\uparrow}0 = \{p(a), q(a), q(b)\} \\
&T_P{\uparrow}1 = \{p(a), q(a), q(b), p(f(a))\} \\
&T_P{\uparrow}2 = \{p(a), q(a), q(b), p(f(a)), p(f(f(a)))\} \\
&\ldots \\
&T_P{\uparrow}\omega = H
\end{aligned}
$$

“Bottom-up” Abstract Interpretation

- Finds an approximation of H by approximating $\mathit{lfp}\ (T_P)$
- We apply abstract interpretation:
  - Domain: I^α, s.t. elements of I^α approximate elements of $I = \wp(B)$.
  - Concretization function: $\gamma : I^\alpha \to I$
  - Abstraction function: $\alpha : I \to I^\alpha$
  - Operator abstraction: abstract version of the T_P operator $T_P^\alpha : I^\alpha \to I^\alpha$
  - Correctness:
    - $(I^\alpha, \gamma, I, \alpha)$ should be a Galois insertion, i.e., , I^α complete lattice and it should approximate I: $\forall M \in I, \gamma(\alpha(M)) \supseteq M$
    - T_P^α safe approximation of T_P, i.e., $\forall d, d \in I^\alpha, \gamma(T_P^\alpha(d)) \supseteq T_P(\gamma(d))$
  - Termination:
    - T_P^α monotonic.
    - I^α (at least) ascending chain finite.
- Then, $H^\alpha = \mathit{lfp}\ (T_P^\alpha) = T_P^\alpha \uparrow n$ will be obtained in a finite number of steps n and H^α will approximate H.

“Bottom-up” Abstract Interpretation (Contd.)



Such “bottom-up” analyses have been proposed for example by Marriott and Sondergaard; Codish, Dams, and Yardeni; Debray and Ramakrishnan; Barbuti, Giacobazzi, and Levi; and others.

Example: simple “type” inference

- Minimal “type inferencing” problem [Sondergaard]:
  Approximating which predicates are in H (“reachability”)
- $pred(a)$: denotes the predicate symbol for an atom a
- $B^\alpha = S$ (set of predicate symbols in a program P)
  Then $I^\alpha = \wp(S)$, we call it S^*
- Concretization function:
  $\gamma : S^* \to I$
  $\gamma(D) = \{a \in B \mid pred(a) \in D\}$
- Abstraction function:
  $\alpha : I \to S^*$
  $\alpha(M) = \{p \in S \mid \exists a \in M, pred(a) = p\}$
- (S^*, γ, I, α) is a Galois insertion

Example: simple "type" inference (Contd.)

- Abstract version of T_P (after some simplification):
  $T_P^\alpha : S^* \rightarrow S^*$

  $$\begin{aligned} T_P^\alpha(D) = \{\, p \in S \mid \exists C \in P,\ & \\ & C = h \leftarrow b_1, \ldots, b_n, \\ & pred(h) \leftarrow pred(b_1), \ldots, pred(b_n) \equiv p \leftarrow p_1, \ldots, p_n, \\ & \text{and } p_1, \ldots, p_n \in D\} \end{aligned}$$

- S^* finite (finite number of predicate symbols in program) and T_P^α monotonic
  $\rightarrow$
  analysis will terminate in a finite number of steps n and
  $H^\alpha = T_P^\alpha \uparrow n$ approximates H.

Example: simple “type” inference (Contd.)

- Example:

$$P = \{\ p(f(X)) \leftarrow p(X). \qquad P_\alpha = \{\ p \leftarrow p.$$
$$p(a). \qquad p.$$
$$r(X) \leftarrow t(X, Y). \qquad r \leftarrow t.$$
$$q(a). \qquad q.$$
$$q(b).\ \} \qquad \}$$

  - ◇ $S = \{p/1, q/1, r/1, t/2\}$
  - ◇ Abstraction:
    $\alpha(\{p(a), p(b), q(a)\}) = \{p/1, q/1\}$
  - ◇ Concretization:
    $$\begin{aligned}\gamma(\{p/1, q/1\}) &= \{A \in B \mid pred(A) = p/1 \vee pred(A) = q/1\} \\ &= \{p(a), p(b), p(f(a)), p(f(b)), \ldots, q(a), q(b), q(f(a)), \ldots\}\end{aligned}$$
  - ◇ Analysis:
    $T_P^\alpha\!\uparrow\!0 = T_P^\alpha(\emptyset) = \{p/1, q/1\}$
    $T_P^\alpha\!\uparrow\!1 = T_P^\alpha(\{p/1, q/1\}) = \{p/1, q/1\} = T_P^\alpha\!\uparrow\!0 = H^\alpha$

T_P-based Bottom-up Analysis: Discussion

- Advantages:
  - ◇ Simple and elegant. Based on the declarative, fixpoint semantics
  - ◇ General: results independent of the query form
- Disadvantages:
  - ◇ Information only about “procedure exit.” Normally information needed at various program points in compilation, e.g., “call patterns” (closures)
  - ◇ The “logical variable” (a.k.a, pointers) not observed (uses ground data). Information on instantiation state, substitutions, aliasing, etc. often needed in compilation
  - ◇ Not query-directed: analyzes whole program, not the part (and modes) that correspond to “normal” use (expressed through a query form)

T_P-based Bottom-up Analysis: Discussion (II)

- Solutions:
  - ◇ Call patterns obtainable via “magic sets” transformation [Marriott and Sondergaard]
    Used also for query-directed analysis by [Barbuti et al.], [Codish et al.], [Gallagher et al.], [Ramakrishnan et al.], and others
  - ◇ Enhanced fixpoint semantics (e.g, S-semantics [Falaschi et al.], [Gaifman and Shapiro])
  - ◇ **Performing top-down analysis**

“Top-down” analysis (summarized)

- Define an extended (collecting) concrete semantics, derived from SLD resolution, making relevant information *observable*.
- Abstract domain: generally “abstract substitutions”.
- Abstract operations: unification, composition, projection, extension, ...
- Abstract semantic function: takes a query form (abstraction of initial goal or set of initial goals) and the program and returns abstract descriptions of the substitutions at relevant program points.
- Variables complicate things:
  - ◇ correctness (due to aliasing),
  - ◇ termination (merging information related to different renamings of a variable)
- Logic variables are in fact (well behaved) pointers:
  `X = tree(N,L,R), L = nil, Y = N, Y = 3, ...`
  this makes analysis of logic programs very interesting
  (and quite relevant to other paradigms).

Domains

- Simple domains [Mellish,Debray], e.g.:
  $\{$ **g**round, **d**on't know, **e**mpty, **f**ree, **n**on-**v**ar $\}$
  (e.g., $f(a)$, ?, $\perp$, X, $f(X)$)
- May need to be very imprecise to be correct:

```
:- entry p(X,Y) : ( var(X), var(Y) ).
p(X,Y) :-
     q(X,Y),
     X = a.
q(Z,Z).
```

  this is the classic pointer aliasing problem!

- Correct/more accurate treatment of aliasing [Debray]:
  associate with a program variable a pair
  $<$ *abstraction of the set of terms the variable may be bound to* ,
  set of program variables it may "share" with $>$.

Domains: Pair Sharing

- More accurate sharing – pair sharing [Sondergaard] [Codish]: pairs of variables denoting possible sharing.

```
:- entry p(X,Y) : ( var(X), var(Y) ).
p(X,Y) :-
     q(X,Y), % { X=f, Y=f } and { (X,Y) }
     X = a.  % { X=g, Y=g } and { (X,Y) }
q(Z,Z).
```

- Note: we have used a “combined” domain: simple modes plus pair sharing
- Pair sharing can encode linearity: (x, x)

```
:- entry p(X,Y) : ( var(X), var(Y) ).
p(X,Y) :-
     q(X,Y),       % { X=f, Y=f } and { (X,Y) }
     W = f(X,Y).   % { W=nv, X=f, Y=f } and { (W,W), (X,Y) }
q(Z,Z).
```

Domains: Set Sharing

- Even more accurate sharing – set sharing [Jacobs et al.] [Muthukumar et al.]: sets of sets of variables.
  $\theta = \{W/a, X/f(A_1, A_2, A_3), Y/g(A_2), Z/A_3\}$
  $\theta^\alpha = \{\emptyset, \{X\}, \{X, Y\}, \{X, Z\}\}$
- A bit tricky to understand. Try:

$$\begin{array}{ccc} \{X\} & \{X,Y\} & \{X,Z\} \\ A_1 & A_2 & A_3 \end{array}$$

$\theta = \{W/a, X/f(A_1, A_2, A_3, B_1), Y/g(h(A_2, B_1)), Z/A_3\}$
$\theta^\alpha = \{\emptyset, \{X\}, \{X, Y\}, \{X, Z\}\}$

$$\begin{array}{ccc} \{X\} & \{X,Y\} & \{X,Z\} \\ A_1 & A_2 + B_1 & A_3 \end{array}$$

- Encodes grounding and independence
  - ◇ W has no ocurrence in any set: it is ground
  - ◇ $\{Y, Z\}$ has no ocurrence in any set: they are independent

Many other domains

- Sharing+Freeness [Muthukumar et al.] (and + depth-K)
- Type graphs [Janssens et al.] [Vuacheret and Bueno, eterms]
- Depth-K [Sato and Tamaki]
- Pattern structure [Van Hentenryck et al.]
- Variable dereferencing [VanRoy] [Taylor]
- ...
- Plus all the work on numerical domains (intervals [Halbwachs,Cousot], polyhedra, octagons, floating point, ...), arrays, etc.
- Much work by [Codish et al.] [File et al.] [Giacobazzi et al.] ... on combining and comparing these domains

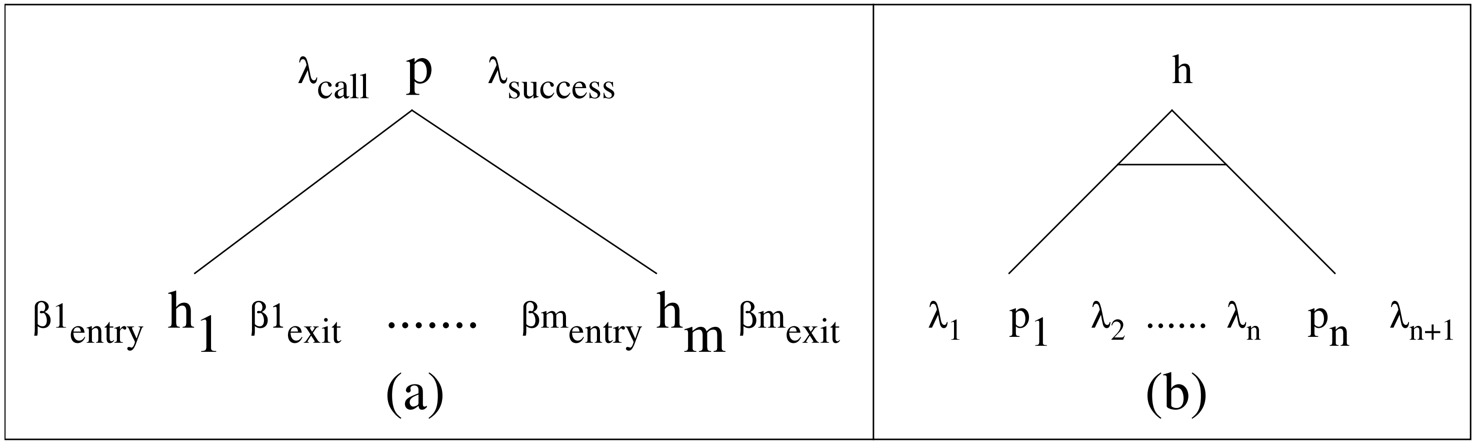
Frameworks

- Predicate level mode inference (call and success patterns for predicates). Unification reformulated as entry + exit unification. Termination by tabling. [Debray et al.]
- Bruynooghe:
  - ◇ Concrete semantics constructs "generalized" AND trees: nodes contain instance of goal before and after execution: *call substitution* and *success substitution*.
  - ◇ Analysis constructs "abstract AND-OR trees". Each represents a (possibly infinite) set of (possibly infinite) concrete trees. Widening to regular trees for termination.
- Muthukumar and Hermenegildo: "PLAI" (the "**top-down algorithm**.") Improvement over previous frameworks: Efficient fixpoint algorithms (dependency tracking) and memory savings (no explicit representation of trees).

Framework is generic: parametric on some basic domain related functions + conditions for correctness and termination.

Abstract AND-OR Tree

- Tree exploration: `?- p.` `h:- p1,...,pn.`

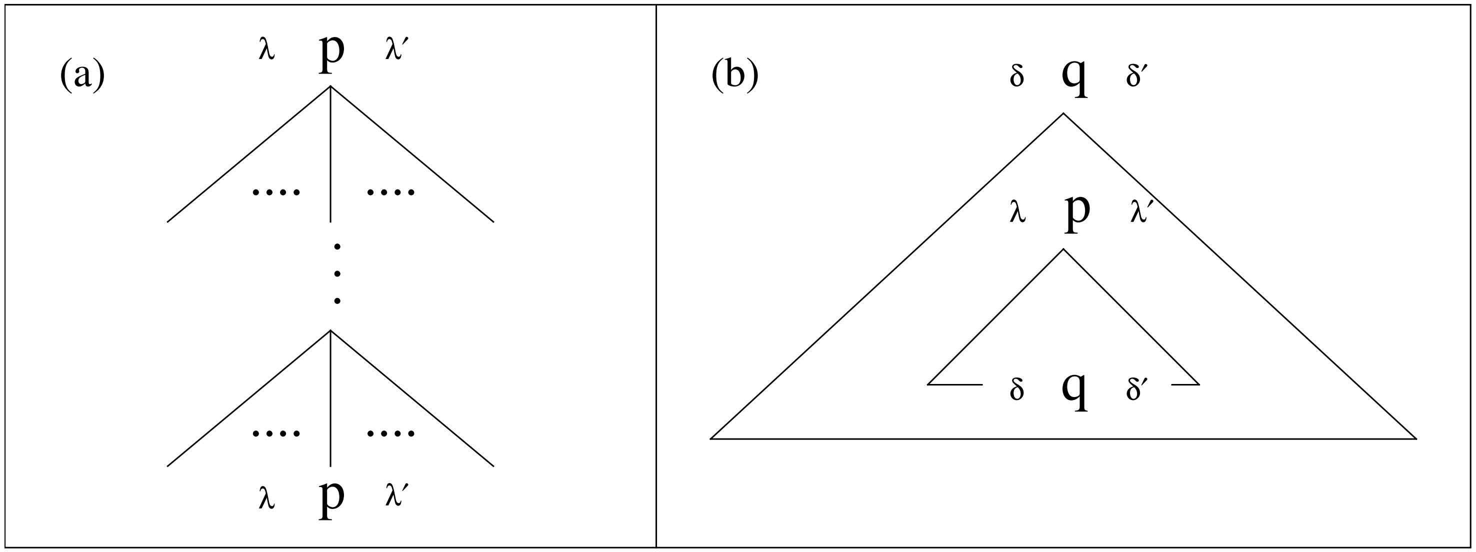


λcall p λsuccess
β1entry h1 β1exit .... βmentry hm βmexit
(a)
h
λ1 p1 λ2 ...... λn pn λn+1
(b)

- Basic operations:
  - ◇ Procedure entry: from λ_{call} obtain $\beta 1_{entry}$
  - ◇ Entry-to-exit `(b)`: from $\beta 1_{entry}$ obtain $\beta 1_{exit}$
  - ◇ Clause entry: from $\beta 1_{entry}$ obtain λ_1 (and clause exit)
  - ◇ Body traversal: from λ_1 obtain λ_{n+1} (iteratively applying `(a)`)
  - ◇ Procedure exit: from (each or all of the) $\beta_{i_{exit}}$ obtain $\lambda_{success}$

Fixpoint Optimization

- Fixpoint required on recursive predicates only:



- Simply recursive `(a)`
- Mutually recursive `(b)`

“Use current success substitution and iterate until a fixpoint is reached”

Analysis of Constraint Logic Programs

- CLP: (relation-based) programs over symbolic and non symbolic domains: constraint satisfaction instead unification (e.g. CLP(R), PrologIII, CHIP, etc.)
- Jorgensen, Marriott, and Michaylov [ISLP'91] and later Marriott and Stuckey [POPL'93] identified numerous opportunities for improvement via static analysis
- A number of proposals for analysis frameworks:
  - ◇ Marriott and Sondergaard [NACLP90]:
    denotational approach
  - ◇ Codognet and Filé [ICPL92]:
    uses constraint solving for the analysis itself and "abstract compilation"
  - ◇ G. de la Banda and Hermenegildo [WICLP'91,ILPS'93]:
    Show that specialized frameworks are not necessary and **LP frameworks (and PLAI in particular) can be used**.

Analysis of Constraint Logic Programs (Contd.)

- Example: Definiteness analysis (Def) [G. de la Banda et al.]
  Domain: $Def = \{d, \wp(\wp(Pvar)), \top\})$

$$\begin{array}{ll}
X = Y + Z & \Rightarrow [(X, [[Y, Z]]), (Y, [[X, Z]]), (Z, [[X, Y]])] \\
X = f(Y, Z) & \Rightarrow [(X, [[Y, Z]]), (Y, [[X]]), (Z, [[X]])] \\
X :: N & \Rightarrow [(X, \top), (N, [[X]])] \\
X > Y & \Rightarrow [(X, \top), (Y, \top)] \\
X = 3 & \Rightarrow [(X, d)]
\end{array}$$

- Other analyses:
  - ◇ Freeness analysis [Dumortier et al.] and combinations.
  - ◇ LSign [Marriott, Sondergaard and Stuckey, ILPS'94]
- Applications:
  - ◇ optimization [Keely et al., CP'96]
  - ◇ parallelization [Bueno et al., PLILP'96]
  - ◇ ...

Origins (Declarative Paradigms, to CLP)

- A few milestones (on the road to CLP analysis):
  - ◇ 1981, Mycroft: strictness analysis of applicative languages
  - ◇ 1981, Mellish: proposes application to logic programs
  - ◇ 1986, Debray: framework with safe treatment of logic variables, discussion of efficiency
  - ◇ 1987, Bruynooghe: framework for LP based on and-or trees
  - ◇ 1987, Jones and Sondergaard: framework based on a denotational definition of SLD
  - ◇ 1988, Warren, Debray and Hermenegildo: Ms and MA^3 practicality of Abs. Int. for Logic Programs shown (for program parallelization). Abstract compilation.
  - ◇ 1989, Muthukumar, Hermenegildo: PLAI framew. (the “top-down algorithm”).
  - ◇ 1990, Van Roy / Taylor: application to sequential optimization of Prolog
  - ◇ 1991, Marriott et al.: first extension to CLP
  - ◇ 1992, Garcia de la Banda and Hermenegildo: generalization of Bruynooghe’s algorithm to CLP, extension of PLAI

Conclusions

- Abstract Interpretation is a very elegant program analysis technique
- It has in addition been proved useful and efficient. E.g., for LP and CLP:
  - ◇ Parallelization of logic (and CLP) programs [Hermenegildo et al]
  - ◇ (Sequential) program optimization [Taylor, VanRoy, ...]
  - ◇ Optimization of CLP programs [Marriott et al, ...]
  - ◇ Abstract debugging, etc.
- Demo!