

ABSTRACT INTERPRETATION

1 BASIC PRINCIPLES

1.1 Introduction

abstractions: widely used concept

some well known examples:

- nine “proof”

$$\begin{array}{rcl}
 & 43 & \xrightarrow{\text{mod}_9} 7 \\
 \times & 78 & \xrightarrow{\text{mod}_9} \overline{\times} 6 \\
 & \text{---} & \text{---} \\
 & 3354 & \xrightarrow{\text{mod}_9} 6
 \end{array}$$

7 is abstraction of 7,16,25,34,43,...

6 is abstraction of 6,15,...,78,...

$7 \overline{\times} 6 = 6$ predicts that the result is one of 6,15,24,...,3354,...

- check dimensions

area of a circle: dimension L^2

which formula is the right one? $2\pi r$ or πr^2

DEFINITION: exact abstraction

$$f : D_1 \times D_2 \times \dots \times D_n \rightarrow D$$

f is a function over concrete domains

$$\alpha_i : D_i \rightarrow D_i^\alpha (i = 1, \dots, n)$$

$$\alpha : D \rightarrow D^\alpha$$

$\alpha_1, \dots, \alpha_n, \alpha$ are abstraction functions, mapping elements from a concrete domain to an abstract domain

$$f^\alpha : D_1^{\alpha_1} \times D_2^{\alpha_2} \times \dots \times D_n^{\alpha_n} \rightarrow D^\alpha$$

f^α is a function over abstract domains

f^α is the *exact* abstraction of f **iff**

for all $j_1 \in D_1, \dots, j_n \in D_n$ holds:

$$\alpha(f(j_1, \dots, j_n)) = f^\alpha(\alpha_1(j_1), \dots, \alpha_n(j_n))$$

first abstraction of arguments then abstract function gives *same* result as first the concrete function then abstraction of result

concrete function: $mult : \mathcal{N} \times \mathcal{N} \rightarrow \mathcal{N}$

$$mult(x, y) = x * y$$

all abstraction functions: $mod_9 : \mathcal{N} \rightarrow \{0, \dots, 8\}$

abstract function:

$$mult^\alpha : \{0, \dots, 8\} \times \{0, \dots, 8\} \rightarrow \{0, \dots, 8\}$$

$$mult^\alpha(x, y) = mod_9(x * y)$$

is exact abstraction

not all functions have an exact abstraction
e.g. $\alpha : \mathcal{Z} \rightarrow \{+, -\}$ $+ \overline{+} - = ?$

some concrete functions are partial
e.g. division by 0 is undefined, division by 9
($\text{mod}_9(9) = 0$) is defined
concrete domain can be extended with \perp

we need a weaker notion

DEFINITION: approximation
give a set S equipped with a partial order \leq
 e_1 approximates e_2 iff $e_2 \leq e_1$

DEFINITION: (safe) abstraction

$$f : D_1 \times D_2 \times \dots \times D_n \rightarrow D$$

f is a *total* function over concrete domains

$$\alpha_i : D_i \rightarrow D_i^{\alpha_i} (i = 1, \dots, n)$$

$$\alpha : D \rightarrow D^\alpha$$

$$f^\alpha : D_1^{\alpha_1} \times D_2^{\alpha_2} \times \dots \times D_n^{\alpha_n} \rightarrow D^\alpha$$

f^α is a *total* function over abstract domains

D^α is equipped with a partial order \sqsubseteq^α

f^α is the (safe) abstraction of f **iff**

for all $j_1 \in D_1, \dots, j_n \in D_n$ holds:

$$\alpha(f(j_1, \dots, j_n)) \sqsubseteq^\alpha f^\alpha(\alpha_1(j_1), \dots, \alpha_n(j_n))$$

EXAMPLE: addition

concrete domain: \mathcal{Z}

concrete function: $add : \mathcal{Z} \times \mathcal{Z} \rightarrow \mathcal{Z}$
 $add(x, y) = x + y$

abstract domain $\mathcal{Z}^\alpha = \{+, zero, -, \top\}$
partial order: $+ \sqsubset^\alpha \top, zero \sqsubset^\alpha \top, - \sqsubset^\alpha \top$

abstraction function $sign : \mathcal{Z} \rightarrow \mathcal{Z}^\alpha$
 $sign(x) =$ if $x > 0$ then $+$
else if $x = 0$ then $zero$
else $-$

$add^\alpha : \mathcal{Z}^\alpha \times \mathcal{Z}^\alpha \rightarrow \mathcal{Z}^\alpha$

add^α	$+$	$zero$	$-$	\top
$+$	$+$	$+$	\top	\top
$zero$	$+$	$zero$	$-$	\top
$-$	\top	$-$	$-$	\top
\top	\top	\top	\top	\top

add^α is a safe abstraction of add

more elaborate EXAMPLE

reduction step in Martelli-Montanari unification

equation system: a multiset of equations over terms

notation $e :: Eqs$ stands for $\{e\} \cup Eqs$

concrete domain: \mathcal{Eqs} the set of concrete equation systems extended with \perp (failure)

concrete function: $f : \mathcal{Eqs} \rightarrow \mathcal{Eqs}$

$$x = x :: Eqs \xrightarrow{\text{remove}} Eqs$$

$$x = t :: Eqs \xrightarrow{\text{substitute}}$$

if $x \in \text{Var}(t)$ then \perp

else $x = t :: Eqs\{x/t\}$ perhaps no change

$$f(t_1, \dots, t_n) = x :: Eqs \xrightarrow{\text{switch}}$$

$$x = f(t_1, \dots, t_n) :: Eqs$$

$$f(t_1, \dots, t_n) = g(s_1, \dots, s_m) :: Eqs \xrightarrow{\text{peel}}$$

if $f/n = g/m$

then $t_1 = s_1 :: \dots :: t_n = s_n :: Eqs$

else \perp

unification: while f can cause a change, apply f

abstract terms:

a variable is an abstract term

\mathcal{G} is an abstract term (the only constant)

if t_1, \dots, t_n are abstract terms (at least one $\neq \mathcal{G}$) then $f(t_1, \dots, t_n)$ is an abstract term

abstract equation system: system of equations over abstract terms

\mathcal{AEqs} : set of abstract equation systems extended with \perp^α

abstraction function $\alpha^t : \text{terms} \rightarrow \text{abstract terms}$

$\alpha^t(t) =$ if $\text{ground}(t)$ then \mathcal{G}
else if $t = f(t_1, \dots, t_n)$
then $f(\alpha^t(t_1), \dots, \alpha^t(t_n))$
else t

can be extended into abstraction function for equation systems:

$\alpha : \mathcal{Eqs} \cup \mathcal{AEqs} \rightarrow \mathcal{AEqs}$

$\alpha(\perp) = \perp^\alpha$

$\alpha(\emptyset) = (\mathcal{G} = \mathcal{G})$

$\alpha(t = s :: \mathcal{Eqs}) =$ if $\text{ground}(t = s)$ then $\alpha(\mathcal{Eqs})$
else $\alpha^t(t) = \alpha^t(s) :: \alpha(\mathcal{Eqs})$

abstract function $f^\alpha : \mathcal{AEqs} \rightarrow \mathcal{AEqs}$

$x = x :: \mathcal{AEqs} \xrightarrow{\text{remove}} \mathcal{AEqs}$

$x = t :: \mathcal{AEqs} \xrightarrow{\text{substitute}}$

if $x \in \text{Var}(t)$ then \perp^α

else $x = t :: \alpha(\mathcal{AEqs}\{x/t\})$ perhaps no change

$f(t_1, \dots, t_n) = x :: \mathcal{AEqs} \xrightarrow{\text{switch}}$

$x = f(t_1, \dots, t_n) :: \mathcal{AEqs}$

$f(t_1, \dots, t_n) = g(s_1, \dots, s_m) :: \mathcal{AEqs} \xrightarrow{\text{peel}}$

if $f/n = g/m = \mathcal{G}/0$ then no change

else if $f/n = \mathcal{G}/0$

then $\alpha(\mathcal{G} = s_1 :: \dots :: \mathcal{G} = s_m :: \mathcal{AEqs})$

else if $g/m = \mathcal{G}/0$

then $\alpha(t_1 = \mathcal{G} :: \dots :: t_n = \mathcal{G} :: \mathcal{AEqs})$

else if $f/n = g/m$

then $\alpha(t_1 = s_1 :: \dots :: t_n = s_n :: \mathcal{AEqs})$

else \perp^α

with partial order $\sqsubseteq^\alpha : \forall \mathcal{AEqs} \neq \perp^\alpha : \perp^\alpha \sqsubseteq^\alpha \mathcal{AEqs}$

if $e :: \mathcal{Eqs} \xrightarrow{f} \mathcal{Eqs}'$ and $ae :: \mathcal{AEqs} \xrightarrow{f^\alpha} \mathcal{AEqs}'$

and $ae = \alpha(e)$ and $ae :: \mathcal{AEqs} = \alpha(e :: \mathcal{Eqs})$

then \mathcal{AEqs}' approximates $\alpha(\mathcal{Eqs}')$

i.e. f^α abstracts f .

Moreover, when $\mathcal{Eqs}' \neq \perp$ then $\mathcal{AEqs}' = \alpha(\mathcal{Eqs}')$.

“While f^α can cause a change, apply f^α ” abstracts Martelli-Montanari unification

Exercise

1. Using the “ground” abstraction of terms, abstract, the following system of equations and perform abstract unification:

$$X = f(a,U),$$

$$Y = f(V,b),$$

$$X = Y,$$

$$g(U,W) = g(V,h(Z)).$$

How close does it mimick concrete unification?

2. Given the append program:

$$\text{append}([], X, X).$$

$$\text{append}([X|U], V, [X|W]) \leftarrow \text{append}(U, V, W).$$

construct an SLD tree for the abstract query $\leftarrow \text{append}(\mathcal{G}, \mathcal{G}, V)$, using abstract unification instead of concrete unification. Which abstract answers do you obtain?

Abstracting Program Behaviour

- program: a finite number of different operations:

program state \rightarrow program state

we can abstract them:

abstract program state \rightarrow

abstract program state

- problem: unbounded sequences through recursion or iteration

also nonterminating program should be abstractable

- problem: many (unbounded) different inputs

one cannot abstract the program behaviour for each separate input

- solution: lift the program behaviour to a function:

SET of program states \rightarrow *SET* of program states

finite number of program points: each point an associated abstract operation

(relevant part of) state: tuples

$$\text{mult} : \mathcal{Z}^2 \rightarrow \mathcal{Z}$$

$$\text{mult}(\langle x, y \rangle) = \langle x * y \rangle$$

$$\text{lifted: } \mathcal{L}\text{mult} : \mathcal{P}(\mathcal{Z}^2) \rightarrow \mathcal{P}(\mathcal{Z})$$

$$\mathcal{L}\text{mult}(S) = \{\langle z \rangle \mid \langle x, y \rangle \in S, \langle z \rangle = \text{mult}(\langle x, y \rangle)\}$$

abstract domain: $\mathcal{Z}^\alpha = \{+, \text{zero}, -\}$

lifted: $\mathcal{P}(\mathcal{Z}^\alpha)$

$$= \{S \mid S \subseteq \mathcal{Z}^\alpha\} = \{\emptyset, \{\langle + \rangle\}, \dots, \{\langle + \rangle, \langle - \rangle, \langle \text{zero} \rangle\}\}$$

$\alpha : \mathcal{Z} \rightarrow \mathcal{Z}^\alpha$ abstraction function

$\alpha(z) =$ if $z > 0$ then $+$

else if $z = 0$ then zero else $-$

$$\alpha^n : \mathcal{Z}^n \rightarrow \mathcal{Z}^{\alpha^n}$$

$$\alpha^n(\langle z_1, \dots, z_n \rangle) = \langle \alpha(z_1), \dots, \alpha(z_n) \rangle$$

$\mathcal{L}\alpha : \mathcal{P}(\mathcal{Z}) \rightarrow \mathcal{P}(\mathcal{Z}^\alpha)$ lifted

$$\mathcal{L}\alpha(S) = \{y \mid x \in S, y = \alpha(x)\}$$

$$\mathcal{L}\alpha^n(S) = \{\bar{y} \mid \bar{x} \in S, \bar{y} = \alpha^n(\bar{x})\}$$

abstract multiplication on the lifted domain

$$\text{Mult}^\alpha : \mathcal{P}(\mathcal{Z}^{\alpha^2}) \rightarrow \mathcal{P}(\mathcal{Z}^\alpha)$$

$$\text{Mult}^\alpha(S) = \{\langle z \rangle \mid \langle x, y \rangle \in S, \langle z \rangle = \text{mult}^\alpha(\langle x, y \rangle)\}$$

where $\text{mult}^\alpha(\langle x, y \rangle)$ abstraction of mult

$$\text{e.g. } \text{Mult}^\alpha(\{\langle +, + \rangle, \langle +, - \rangle\}) = \{\langle + \rangle, \langle - \rangle\}$$

Observations

- lifted concrete domain \mathcal{D} : sets of states ordered by subset relation
- lifted abstract domain \mathcal{A} : sets of abstract states ordered by subset relation
- \mathcal{D} and \mathcal{A} are complete lattices top element: the original domain of concrete/abstract states
e.g. $\{+, zero, -\}$ more precise $\{\langle + \rangle, \langle zero \rangle, \langle - \rangle\}$ is top of \mathcal{A}
bottom element: the empty set
- every set of states has an abstraction
 $\alpha(\{3, 8, 11\}) = \{+\}$
 $\alpha(\{-3, -8, 0\}) = \{-, zero\}$
- abstractions $a \supseteq \alpha(d)$ are approximations of $\alpha(d)$, the abstraction of d
e.g. $\{+, zero\}$ is an approximation of the abstraction of $\{3, 8, 11\}$

- $a = \alpha(d)$ is an approximation of the abstraction of $d' \subseteq d$
e.g. $\{-, zero\}$ is the abstraction of $\{-3, -8, 0\}$ and approximates the abstraction of $\{-3, -8\}$
- but the best approximation of the abstraction is the abstraction itself
- α is the abstraction function: $\mathcal{D} \rightarrow \mathcal{A}$
 γ is the concretisation function: $\mathcal{A} \rightarrow \mathcal{D}$
 $\gamma(a) =$ the largest set abstracted by $a =$ the union of all sets abstracted by a
e.g. $\gamma(\{+\}) = \{1, 2, 3, \dots\} = \mathcal{Z}^+$
 γ gives the meaning of the abstract elements

BUT abstract domain as powerset: too rigid,
e.g a more concise domain is

$$\{\top = \{+, zero, -\}, \perp = \emptyset\}$$

BUT concrete domain as powerset: too rigid,
e.g. layers of abstraction

→ Galois connection as structure

where did we get, what we want to achieve

- to circumvent problems of infinite inputs and infinite sequences of operations, we collect states of the computation in finite number of program points
- so lifted operations map sets of computation states to sets of computation states
- by choosing appropriate abstract domains, and appropriate abstract operations, we can obtain descriptions of these sets of states
- “appropriate” domains: allowing to express properties of interest
- “appropriate” operations: allowing to obtain “safe” approximations in a finite amount of time
- once appropriate domain and operation chosen and implemented, the analysis of particular programs is automatic, can be done at compile-time
- the “art”: choosing right mix of domain and operations providing the desired info with the desired precision with a minimal amount of analysis time
- domain may need to express not only properties of interest, but also other properties to allow abstract operations to achieve desired level of precision e.g. groundness in LP

1.2 GALOIS CONNECTION

DEFINITION

Let (D, \subseteq) be the concrete domain equipped with a partial order

Let (A, \sqsubseteq) be the abstract domain equipped with a partial order

A Galois connection between A and D is a pair of functions α, γ satisfying:

- α is total from D to A , it is the abstraction function, every concrete element has an abstraction
- γ is total from A to D , it is the concretisation function, every abstract element has a “meaning”
- $\forall a \in A, \forall d \in D : \alpha(d) \sqsubseteq a$ **iff** $d \subseteq \gamma(a)$
 - \rightarrow : if a approximates the abstraction of d then the meaning of a approximates d
 - \leftarrow : if the meaning of a approximates d then a approximates the abstraction of d

$$\begin{array}{ccc} \gamma(a) & \longleftarrow & a \\ \bigvee & \iff & \bigvee \\ d & \longrightarrow & \alpha(d) \end{array}$$

an alternative DEFINITION

Let (D, \subseteq) be the concrete domain equipped with a partial order

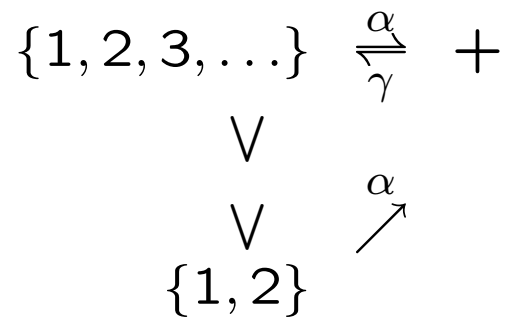
Let (A, \sqsubseteq) be the abstract domain equipped with a partial order

A Galois connection between A and D is a pair of functions α, γ satisfying:

- α is total and monotone from D to A
the bigger an element, the bigger its abstraction
- γ is total and monotone from A to D
the bigger an abstraction, the bigger its meaning
- $\forall a \in A, \forall d \in D :$
 $d \subseteq \gamma(\alpha(d))$ the meaning of the abstraction of an element approximates the element, i.e. the loss of information due to abstraction is sound
 $\alpha(\gamma(a)) \sqsubseteq a$ an abstract element approximates the abstraction of its meaning, i.e. γ introduces no loss of information

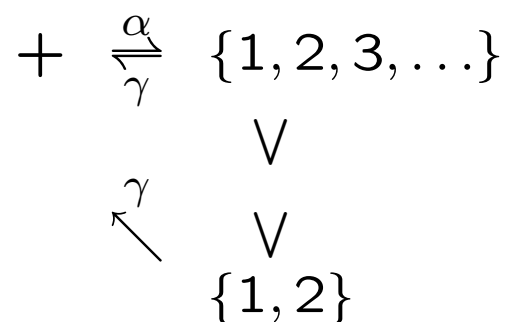


Notice the assymetrie! in general, A and D cannot be switched



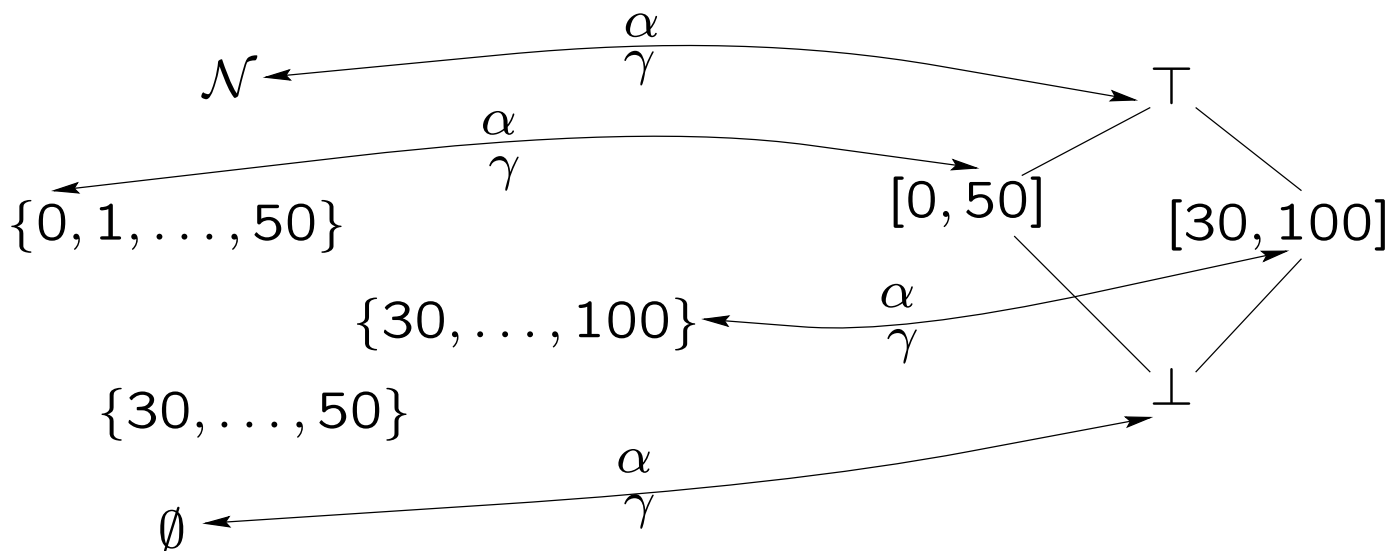
the inverse of α is not a function

Can γ be the abstraction and α the concretisation?



violates: $\alpha(\gamma(\{1, 2\})) \sqsubseteq \{1, 2\}$

need to invert order relation as well



infinite concrete domain, 4-element abstract domain

Galois connection?

what is abstraction of $\{30, \dots, 50\}$

- $[0, 50]$? no, α is not monotonic, consider $\{30, \dots, 51\}$
- $[30, 100]$? no, α is not monotonic, consider $\{29, \dots, 50\}$
- \top ? no, α not monotonic
- \perp ? no, $\{30, \dots, 50\} \not\subseteq \gamma(\alpha(\{30, \dots, 50\}))$

need interval $[30, 50]$ in abstract domain

Two abstract elements need a glb that is an abstraction of the glb of their concretisation.

Properties of Galois connections

- $\gamma(\alpha(\gamma(a))) = \gamma(a)$
no information is lost by abstracting the meaning of a
- $\alpha(\gamma(\alpha(d))) = \alpha(d)$
abstracting the meaning of the abstraction of d gives back the same abstraction, i.e. no additional info is lost by abstracting a second time
- $\alpha(\text{lub}(d_1, \dots, d_n)) = \text{lub}(\alpha(d_1), \dots, \alpha(d_n))$
it makes no difference whether you abstracts the *lub* of a set of concrete elements union of computation states or takes the *lub* of their abstractions e.g sign abstraction $\alpha(\text{lub}(\{1, 3\}, \{0, 7\}))$
lub and γ not necessarily commute
- $\gamma(\text{glb}(a_1, \dots, a_n)) = \text{glb}(\gamma(a_1), \dots, \gamma(a_n))$ it makes no difference whether you take the meaning of the *glb* of a set of abstract elements or take the *glb* intersection of computation states of their meaning e.g sign abstraction $\gamma(\text{glb}(\{+, \text{zero}\}, \{-, \text{zero}\}))$
glb and α not necessarily commute
- given (A, \sqsubseteq) and (D, \subseteq) , α determines γ : $\gamma(a)$ is the *lub* of all elements abstracted by a
- given (A, \sqsubseteq) and (D, \subseteq) , γ determines α : $\alpha(d)$ is the *glb* of all elements whose meaning approximates d

$$\begin{array}{ccc}
 & & a \\
 & \swarrow \gamma & \downarrow \vee \\
 \gamma(a) & \xrightarrow[\gamma]{\alpha} & \alpha(\gamma(a))
 \end{array}$$

$$\alpha(\gamma(a)) \sqsubseteq a$$

it means that a is “garbage”, unneeded element

when no such elements: $\alpha(\gamma(a)) = a$

Galois surjection or Galois insertion

every element of A is abstraction of some d

all elements of A have a different meaning

assumption $d = \gamma(a_1) = \gamma(a_2)$ implies

$$\alpha(\gamma(a_1)) \neq a_1 \text{ or } \alpha(\gamma(a_2)) \neq a_2$$

Galois insertion:

- if (D, \subseteq) fixed, α fixed then \sqsubseteq is implied:
 $\alpha(d_1) \sqsubseteq \alpha(d_2)$ **iff** $d_1 \subseteq d_2$
- similarly if (D, \subseteq) fixed, a feasible A is fixed
and if γ is fixed: $a_1 \sqsubseteq a_2$ **iff** $\gamma(a_1) \subseteq \gamma(a_2)$
- if D is complete lattice, then so is A

1.3 Approximating Semantics

- the “operational” semantics which runs the program and collects in each program point the state, each time control reaches that point, and this for all inputs, is a monotonic continuous function
- a standard approach to semantics is to describe it as the least fixpoint of a monotone continuous function, its value describes the properties of interest of the program
- e.g. the fixpoint of the T_P operator
- the core theory of abstract interpretation developed by P. Cousot and R. Cousot is about how to approximate the fixpoint of a monotone continuous function over a concrete domain D by another function over the abstract domain A , where there is a Galois connection between A and D
- abstract interpretation is not impossible when the function is not continuous, or the relation between A and D is weaker than a Galois connection

How to obtain a (precise) approximation of the abstraction of the least fixpoint of F the semantic function: $F : D \rightarrow D$

The most precise way: $\alpha(lfp(F))$: i.e. compute the least fixpoint and abstract it

$$\begin{array}{c}
 \top \\
 F \downarrow \\
 \vdots \\
 F \downarrow \\
 F \circ \quad gfp(F) \\
 F \uparrow \\
 \vdots \\
 F \circ \quad fp(F) \\
 \vdots \\
 F \circ \quad lfp(F) \xrightarrow{\alpha} \alpha(lfp(F)) \\
 F \uparrow \\
 \vdots \\
 F \uparrow \\
 F \uparrow \\
 \emptyset
 \end{array}$$

but rarely useable, the number of iterations required to reach $lfp(F)$ is typically unbounded

a bit more practical and a bit less precise

define $F^\alpha : A \rightarrow A$

$$F^\alpha(a) = \alpha(F(\gamma(a)))$$

e.g. $p([\])$ ← $p([X|L])$ ← $p(L)$

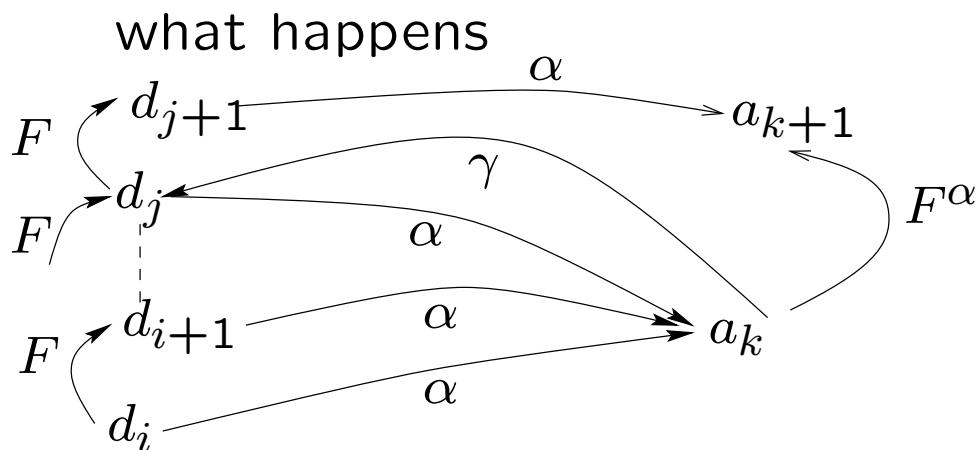
$$\alpha(T_P(\gamma(\perp))) = \alpha(T_P(\emptyset)) = \alpha(p([\])) = p(\text{nil-terminated list})$$

$$\alpha(T_P(\gamma(p(\text{nil-terminated list}))))$$

$$= \alpha(T_P(\text{the set of all } p(\text{nil-terminated list})))$$

$$= \alpha(\text{the set of all } p(\text{nil-terminated list}))$$

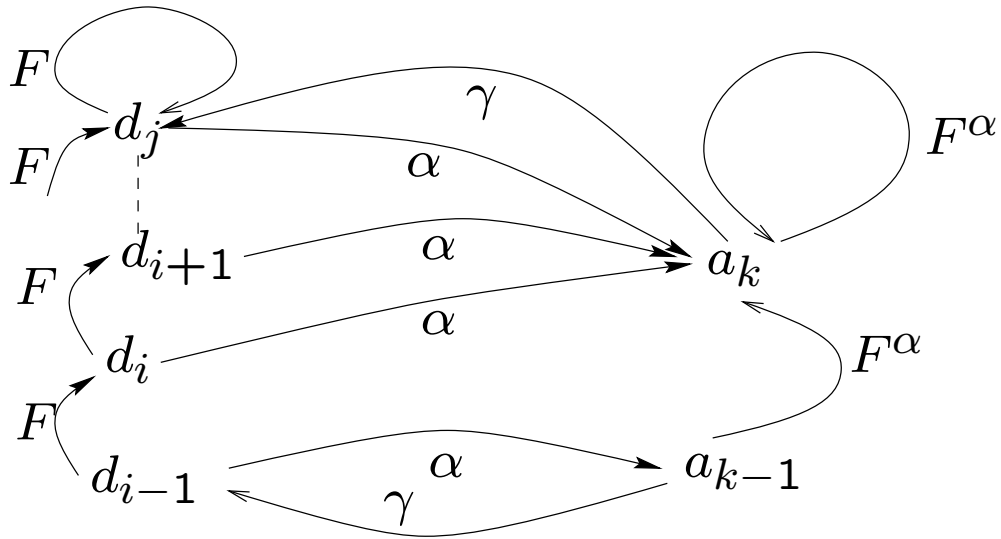
$$= p(\text{nil-terminated list})$$



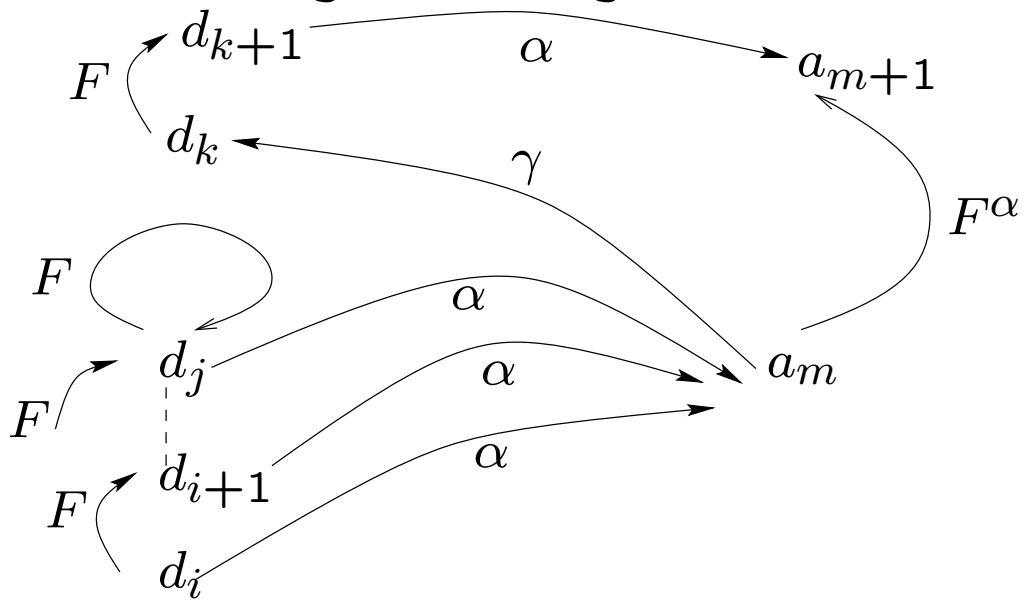
the lifted function F has often to be applied on an infinite set

i.e. the original function f has to be applied an infinite number of times

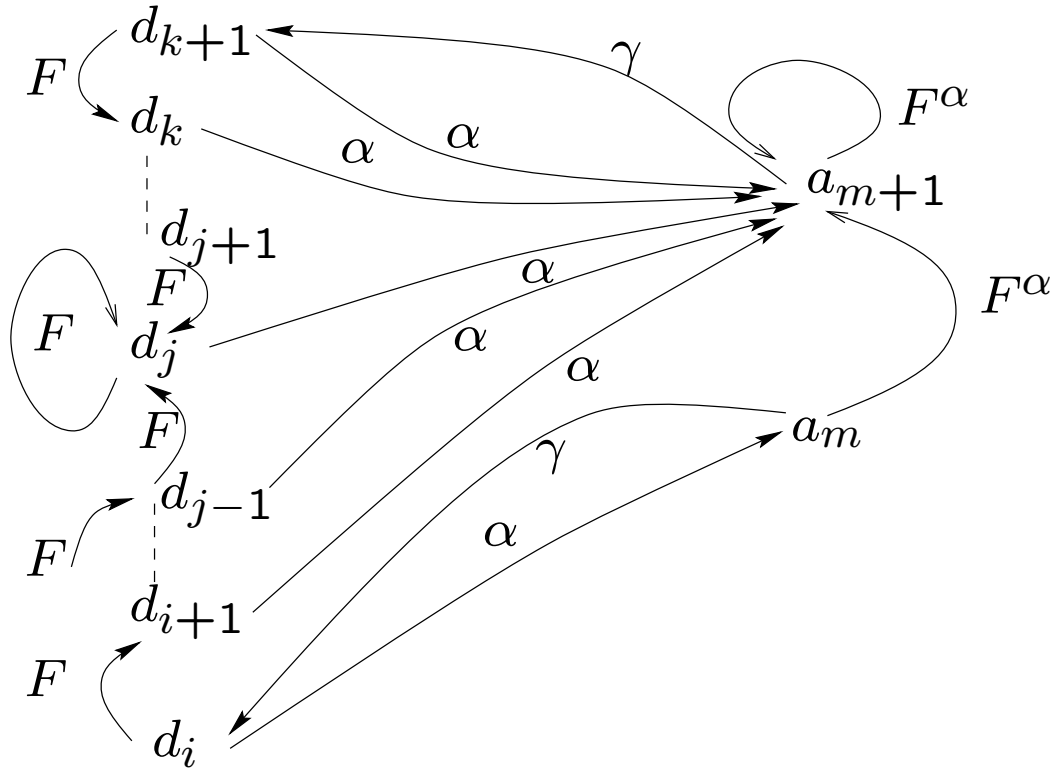
when things go well: a fixpoint



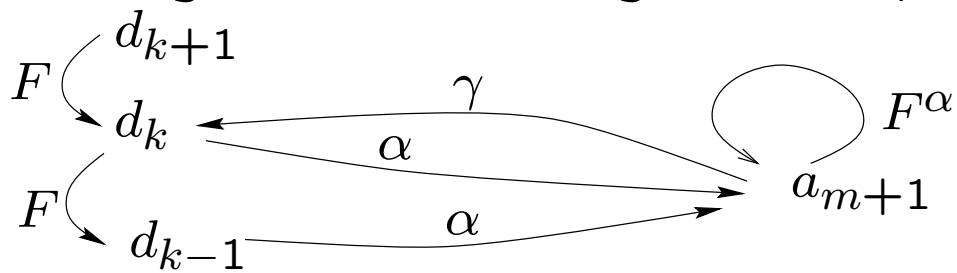
when things do not go well: a missed fixpoint



but the greatest fixpoint of F cannot escape



although F^α can have greater fixpoint



a practical approach:

define $Fp^\alpha : A \rightarrow A$ and prove that

$F^\alpha(a) = \alpha(F(\gamma(a))) \sqsubseteq Fp^\alpha(a)$ for all $a \in A$

ideal: $Fp^\alpha(a) = F^\alpha(a)$ for all $a \in A$

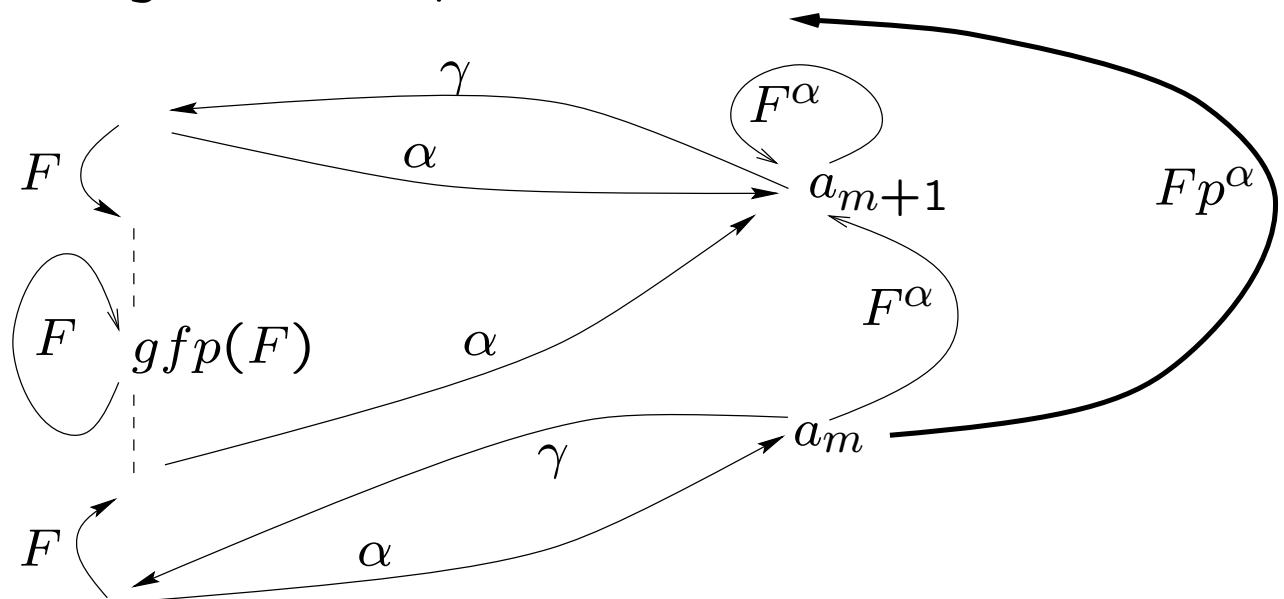
e.g. $\{+\} +^\alpha \{+\} = \{+\}$: ideal

$\{+\} +^\alpha \{+\} = \{+, -\}$: less precise

take care Fp^α is monotone and continuous,
so that its fixpoint can be found/approximated
in finite time

can be arbitrary worse than $\alpha(gfp(F))$

the greatest fixpoint can be missed



how to obtain fixpoint of abstract function?

A finite: terminates always

all chains $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \dots$ finite: terminates always

what if infinite chains?

what if too many iterations?

widening/narrowing

widening:

replace $Fp^\alpha(a)$ by $Fp^\alpha(a) \nabla a \sqsupseteq a$

such that, after finite iterations,

a value a_n is reached for which

$Fp^\alpha(a_n) \sqsubseteq a_n$

(e.g. (drastic) $a_n = \top$)

narrowing:

when $a_n \sqsupseteq gfp(Fp^\alpha)$ then one can iterate until $gfp(Fp^\alpha)$ but one can stop any time, the value is always

an approximation of $gfp(Fp^\alpha)$

2. APPLIED on LOGIC PROGRAMS

2.1 About abstracting T_P semantics

$$T_P : \mathcal{P}(H_B) \rightarrow \mathcal{P}(H_B)$$

e.g. pre-interpretation J as α

T_P^J is the abstract operator:

$$T_P^J(I) = \{p(J(V(t_1)), \dots, J(V(t_n))) : \\ p(t_1, \dots, t_n) \leftarrow B_1, \dots, B_k \in P, \text{ and} \\ I \models_V B_1 \wedge \dots \wedge B_k\}$$

$$p([\]) \leftarrow$$

$$p([X, Y|Z]) \leftarrow p(Z)$$

a pre-interpretation J : $D = \{odd, even\}$ $J([\]) = even$, $J([-|even]) = odd$, $J([-|odd]) = even$

$$T_P^J \uparrow 1 = \{p(J([\]))\} = \{p(even)\}$$

$$T_P^J \uparrow 2 = \{p(even)\} \cup \{p(J([V(X), V(Y)|even]))\} \\ = \{p(even)\} \cup \{p(even)\} = \{p(even)\}$$

is the fixpoint

can be used to prove that query $\leftarrow p([U, V, W])$ cannot give an answer, indeed $p(J(V([U, V, W])))$ is $p(odd)$ for all valuations V

2.1.1. A famous example: PROP/POS

$$D = \{g, \neg g\}$$

$$J(f(t_1, \dots, t_n)) =$$

if $J(t_1) = g$ and ... and $J(t_n) = g$

then g else $\neg g$

the preinterpretation of a ground term is g

$$a([], X, X) \leftarrow$$

$$a([X|U], V, [X|W]) \leftarrow a(U, V, W)$$

$$T_P^J \uparrow 1 = \{a(g, g, g), a(g, \neg g, \neg g)\}$$

$$T_P^J \uparrow 2 = T_P^J \uparrow 1 \cup$$

$$\{a(g, g, g), a(\neg g, g, \neg g), a(g, \neg g, \neg g), a(\neg g, \neg g, \neg g)\}$$

$$T_P^J \uparrow 3 = T_P^J \uparrow 2 =$$

$$\{a(g, g, g), a(g, \neg g, \neg g), a(\neg g, g, \neg g), a(\neg g, \neg g, \neg g)\}$$

queries: $J(a([a, b], [c], X)) = \{a(g, g, g), a(g, g, \neg g)\}$

answer: $\{a(g, g, g)\}$ i.e. X is ground

$J(a(X, Y, [c, d])) = \dots$ answer $\{a(g, g, g)\}$ i.e.

X and Y ground

$J(a([a, b], Y, Z)) = \dots$ answer $\{a(g, g, g), a(g, \neg g, \neg g)\}$

i.e. Y and Z are not ground

Abstract Compilation: apply J on program

$X = []$ becomes $J(X) = J([])$ i.e. solutions of $X = g$

$X = f(Y, Z)$ becomes $J(X) = J(f(Y, Z))$ i.e. solutions of $j2(X, Y, Z)$ with $j2/3$:

$j2(g, g, g) \leftarrow$

$j2(\neg g, g, \neg g) \leftarrow$

$j2(\neg g, \neg g, g) \leftarrow$

$j2(\neg g, \neg g, \neg g) \leftarrow$

$a(g, X, X) \leftarrow$

$a(L1, V, L3) \leftarrow j2(L1, X, U), j2(L3, X, W), a(U, V, W).$

Model of $j2(X, Y, Z)$ is model of “positive” boolean formula $X \leftrightarrow Y \wedge Z$

Positive (POS): evaluates to true for all variables true

Is datalog program, applying T_P gives the model

Can also be used to derive types of answer:
e.g. $J([]) = list, J([-|list]) = list, J(a) = \neg list, J([-|\neg list]) = \neg list, \dots$

Magic Set Transformation: gives call-modes

idea: query becomes fact
define calls and answers

example

query: $append([a, b], [c], L)$.

$call_app([a, b], [c], L)$.

If there is a call that is an instance of $append([], X, X)$
then that instance is also an answer:

$answer_app([], X, X) \leftarrow call_app([], X, X)$.

If there is a call of the form $append([X|U], V, [X|W])\theta$
then there is also a call of the form $append(U, V, W)\theta$:
 $call_app(U, V, W) \leftarrow call_app([X|U], V, [X|W])$.

If there is a call of the form $append([X|U], V, [X|W])\theta$
and an answer of the form $append(U, V, W)\theta\sigma$,
then there is also an answer of the form
 $append([X|U], V, [X|W])\theta\sigma$:

$answer_app([X|U], V, [X|W]) \leftarrow$
 $call_app([X|U], V, [X|W]), answer_app(U, V, W)$.

Exercise

3. For the program

$call_app([a, b], [c], L)$.

$answer_app([], X, X) \leftarrow call_app([], X, X)$.

$call_app(U, V, W) \leftarrow call_app([X|U], V, [X|W])$.

$answer_app([X|U], V, [X|W]) \leftarrow call_app([X|U], V, [X|W]), answer_app(U, V, W)$

- Compute the Least Herbrand Model
- Apply abstract compilation on it (using the $\{g, \neg g\}$ domain)
- Compute the Least Herbrand Model of the abstract program.
What does it tell?

2.1.2 SIZE relations as used in termination analysis

$$size : U_L \rightarrow \mathcal{N}$$

$$size(t) = \begin{cases} \text{if } t = [X|Y] \text{ then } 1 + size(Y) \\ \text{else } 0 \end{cases}$$

basis of an abstraction function:

B_p : the Herbrand base for predicate p/k

\mathcal{N}^k : the set of k -dimensional points

One function for each predicate

$$\alpha_p^{size} : \mathcal{P}(B_p) \rightarrow \mathcal{P}(\mathcal{N}^k)$$

$$\alpha_p^{size}(S) = \{ \langle size(t_1), \dots, size(t_k) \rangle \mid p(t_1, \dots, t_k) \in S \}$$

$$\gamma_p^{size} : \mathcal{P}(\mathcal{N}^k) \rightarrow \mathcal{P}(B_p)$$

$$\gamma_p^{size}(S) = \{ p(t_1, \dots, t_k) \mid \langle size(t_1), \dots, size(t_k) \rangle \in S \}$$

Order relation in $\mathcal{P}(\mathcal{N}^k) : \subseteq$ (as in $\mathcal{P}(B_p)$)

Note: $S_1, S_2 \in \mathcal{N}^k : S_1 \subseteq S_2$ iff $\gamma_p^{size}(S_1) \subseteq$

$$\gamma_p^{size}(S_2)$$

$$\perp = \emptyset, \top = \mathcal{N}^k$$

a Galois connection (still infinite)

Second level of abstraction:

System of linear equations in \mathcal{R}^k describes a set of points, i.e. the solutions

One equation describes a $k - 1$ dimensional hyperplane

A set of equations describes the intersection

Different equations systems can describe the same solution space

consider the equivalence classes

Abstract domain: equivalence classes of systems of equations in k variables: $\mathcal{P}(\mathcal{E}_q^k)$

$$\gamma^{eq} : \mathcal{P}(\mathcal{E}_q^k) \rightarrow \mathcal{P}(\mathcal{N}^k)$$

$$\gamma^{eq}(Eq) = \{ \langle n_1, \dots, n_k \rangle \mid \langle n_1, \dots, n_k \rangle \in \mathcal{N}^k \text{ and } \langle n_1, \dots, n_k \rangle \text{ is a solution of } Eq \}$$

$$\alpha^{eq} : \mathcal{P}(\mathcal{N}^k) \rightarrow \mathcal{E}_q^k$$

$\alpha^{eq}(S) = Eq$ such that $Eq \in \mathcal{P}(\mathcal{E}_q^k)$ and $S \subseteq \gamma^{eq}(Eq)$ and Eq is minimal no Eq' has a smaller solution space

Order: $Eq_1 \sqsubseteq Eq_2$ iff $\gamma^{eq}(Eq_1) \subseteq \gamma^{eq}(Eq_2)$

\perp unsolvable systems, e.g. $\{0 = 1\}$

\top : trivial system, e.g. $\{0 = 0\}$

A Galois connection

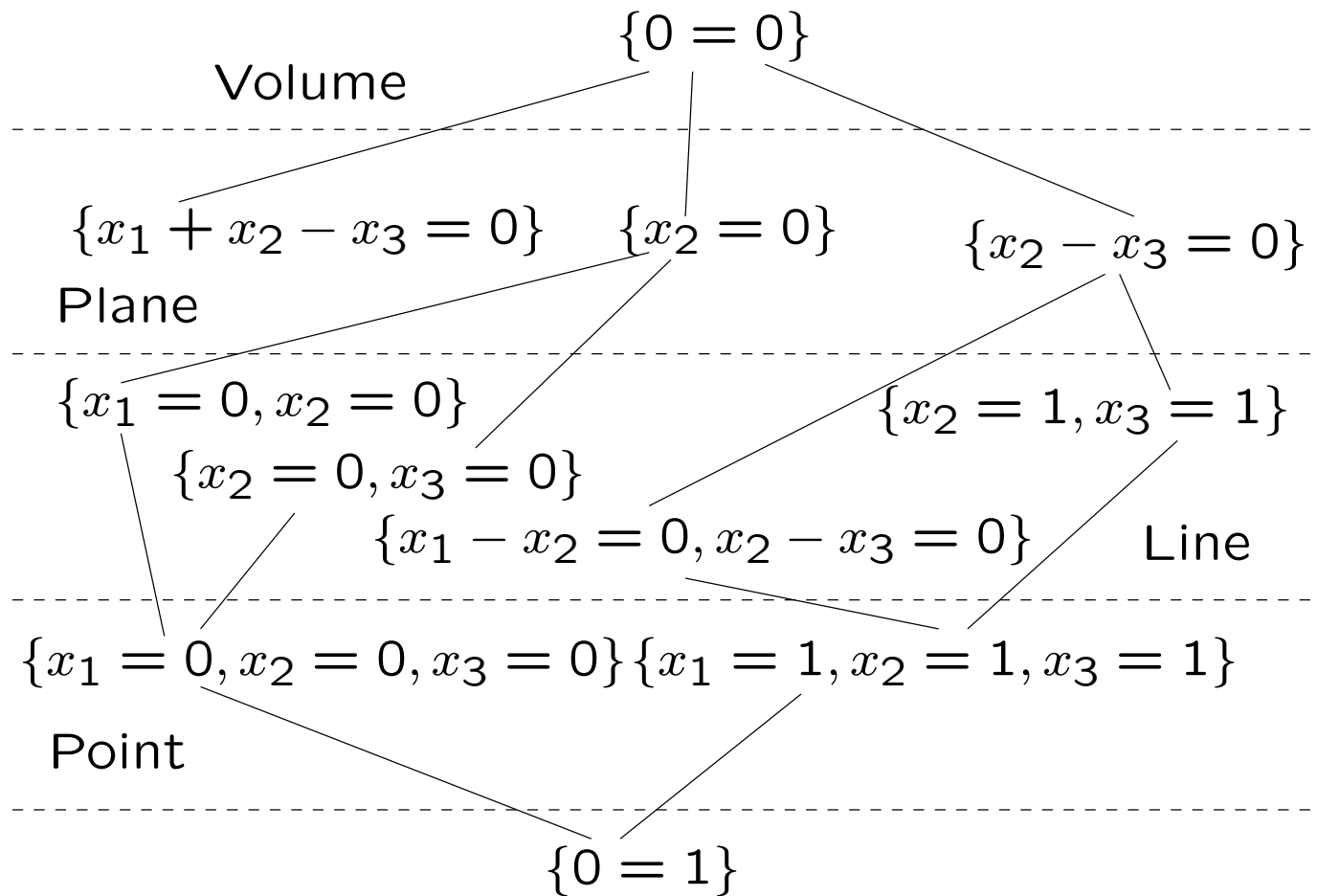
Composition of both abstractions is also a Galois connection

for each predicate p/k : a system of equations in k variables

$$\gamma : \mathcal{P}(\mathcal{E}q^k) \rightarrow \mathcal{P}(B_p)$$

$$\gamma(Eq) = \{p(t_1, \dots, t_k) \mid \langle size(t_1), \dots, size(t_k) \rangle \text{ is a solution of } Eq\}$$

abstract domain for 3-argument predicate



How to compute abstraction of set of atoms?

- Initialise E_q with \perp
- While there is an atom $P(t_1, \dots, t_k)$ such that $\langle size(t_1), \dots, size(t_k) \rangle$ is not a solution of E_q do:
 - Replace E_q with a new system having as solutions all previous solutions and $\langle size(t_1), \dots, size(t_k) \rangle$

$$T_P^\alpha(I) = \alpha(T_P(\gamma(I)))$$

example append

$$a([], X, X) \leftarrow$$

$$a([X|U], V, [X|W]) \leftarrow a(U, V, W)$$

$$T_P^\alpha \uparrow 1 = \alpha(T_P(\gamma(\perp)))$$

$$= \alpha(T_P(\emptyset))$$

$$= \alpha(\{a([], X, X) \mid X \in H_U\})$$

$$= \{x_1 = 0, x_2 = x_3\}: \text{ a line}$$

$$\begin{aligned}
T_P^\alpha \uparrow 2 &= \alpha(T_P(\gamma(\{x_1 = 0, x_2 = x_3\}))) \\
&= \alpha(T_P(\{a(t_1, t_2, t_3) \mid \text{size}(t_1) = 0, \text{size}(t_2) = \\
&\quad \text{size}(t_3)\})) \\
&= \alpha(\{a([], X, X) \mid X \in H_U\} \cup \{a([X|U], V, [X|W]) \mid \\
&\quad X, U, V, W \in H_U \wedge \text{size}(U) = 0, \text{size}(V) = \text{size}(W)\}) \\
&= \text{lub}(\alpha(\{a([], X, X) \mid X \in H_U\}), \alpha(\{a([X|U], V, [X|W]) \\
&\quad \mid X, U, V, W \in H_U \wedge \text{size}(U) = 0, \text{size}(V) = \text{size}(W)\})) \\
&= \text{lub}(\{x_1 = 0, x_2 = x_3\}, \{x_1 = 1, x_2 = x_3 - \\
&\quad 1\}): 2 \text{ parallel lines} \\
&= \{x_1 + x_2 = x_3\}: \text{ a plane}
\end{aligned}$$

$$\begin{aligned}
T_P^\alpha \uparrow 3 &= \alpha(T_P(\gamma(\{x_1 + x_2 = x_3\}))) \\
&= \alpha(T_P(\{a(t_1, t_2, t_3) \mid \text{size}(t_1) + \text{size}(t_2) = \\
&\quad \text{size}(t_3)\})) \\
&= \alpha(\{a([], X, X) \mid X \in H_U\} \cup \{a([X|U], V, [X|W]) \mid \\
&\quad X, U, V, W \in H_U \wedge \text{size}(U) + \text{size}(V) = \text{size}(W)\}) \\
&= \text{lub}(\alpha(\{a([], X, X) \mid X \in H_U\}), \alpha(\{a([X|U], V, [X|W]) \\
&\quad \mid X, U, V, W \in H_U \wedge \text{size}(U) + \text{size}(V) = \text{size}(W)\})) \\
&= \text{lub}(\{x_1 = 0, x_2 = x_3\}, \{x_1 + x_2 = x_3\}) \\
&= \{x_1 + x_2 = x_3\}: \text{ same plane}
\end{aligned}$$

Automation: unification is translated in equations (again a form of abstract compilation)

$$a([], X, X) \leftarrow$$

is translated into:

$$a(x_1, x_2, x_3) \leftarrow x_1 = 0, x_2 = x_3$$

$a([X|U], V, [X|W]) \leftarrow a(U, V, W)$ is translated into:

$$a(x_1, x_2, x_3) \leftarrow x_1 = 1 + u, x_2 = v, \\ x_3 = 1 + w, a(u, v, w)$$

$$T_P^\alpha \uparrow 1 : x_1 = 0, x_2 = x_3$$

Recursive clause:

$$T_P^\alpha \uparrow 2 : x_1 = 1 + u, x_2 = v, x_3 = 1 + w, u = 0, v = w$$

$$\rightsquigarrow x_1 = 1, x_3 = 1 + x_2 \text{ other equations not relevant}$$

$$lub(\{x_1 = 0, x_2 = x_3\}, \{x_1 = 1, x_3 = 1 + x_2\}) = \{x_3 = x_1 + x_2\}$$

$$T_P^\alpha \uparrow 3 : x_1 = 1 + u, x_2 = v, x_3 = 1 + w, w = u + v$$

$$\rightsquigarrow \{x_3 = x_1 + x_2\}$$

Lub of equations: algorithm of Karr (Acta Informatica 1976)

Richer domain: k-dimensional intervals

Now infinite chains possible

e.g. $[0, 1], [0, 2], [0, 3], \dots$

needs widening:

$$[l_1, u_1] \nabla [l_2, u_2] = [l_3, u_3]$$

where $l_3 =$ if $l_1 = l_2$ then l_1 else 0

$$u_3 = \text{if } u_1 = u_2 \text{ then } u_1 \text{ else } \infty$$

Or still richer: convex hulls

Cousot and Halbwachs POPL78

Was used by them to derive non-trivial assertions from imperative programs

$$\text{del_all}(X, [], []) \leftarrow$$

$$\text{del_all}(X, [X|L], R) \leftarrow \text{del_all}(X, L, R)$$

$$\text{del_all}(X, [Y|L], [Y|R]) \leftarrow X \neq Y, \text{del_all}(X, L, R)$$

abstraction: $x_3 \leq x_2$

Exercise

4. Consider the program

$part(X, [], [], []) \leftarrow .$

$part(X, [Y|L], [Y|L1], L2) \leftarrow X \geq Y, part(X, L, L1, L2).$

$part(X, [Y|L], L1, [Y|L2]) \leftarrow X < Y, part(X, L, L1, L2).$

Using `listlength` as `size`, check that $size(L) = size(L1) + size(L2)$ is a fixpoint for the predicate $part(X, L, L1, L2)$ under the abstract T_P operator.

2.2 About abstracting procedural behaviour

difficult without limiting computation rule

2.2.1 Abstracting OLDT

OLD derivations

computation rule always select atom among most recent ones, e.g. left most (LD-derivations)

OLD-tree: as SLD-tree but for OLD computation rule

OLD-semantics: set of OLD-trees

exhibits run-time behaviour

- which atoms are selected
- how is selected atom instantiated
- what is computed answer of selected atom

Useful e.g. for code optimisation: specialisation of unifications

Preferred above “magic sets” for domains not suited for “abstract compilation”

Can be formulated as fixpoint of some function?

as fixpoint of sequence of partial OLD-trees

$F_{P,SG} : \text{partial OLD-tree} \rightarrow \text{partial OLD-tree}$

the trees with as only node a query $\leftarrow Q$ with $\leftarrow Q \in SG$ are partial OLD-trees

$F_{P,SG} \uparrow 0 = \{OLD_p \mid OLD_p \text{ has one node labeled } \leftarrow Q \text{ with } \leftarrow Q \in SG\}$

the *extension* of a partial OLD-tree is a partial OLD-tree

$F_{P,SG} \uparrow (n + 1) = \{OLD_p \mid OLD_p \text{ is the extension of a partial OLD-tree in } F_{P,SG} \uparrow n\}$

extension: process all unmarked leaves different from \square as follows:

if selected atom has no matching clauses:

mark leaf with *failure*

else add all resolvents as children

the fixpoint of $F_{P,SG}$ is the set of complete OLD-trees

it describes the operational behaviour for the set of queries in SG

How to abstract set of OLD-trees?

OLD-tree:

- skeleton:
 - tree structure
 - identifiers of clauses used in resolution step
 - atoms from original program
- adornments
 - indices for the renaming of clause variables
 - substitutions

a straightforward abstraction:

keep skeletons, abstract substitutions

problem with unbounded depth of trees

for good intuition: first modify OLD

OLDT: OLD with Tabulation

processing of $\leftarrow g_i, G$ is modified:

- if there is a table entry for a renaming of g_i then *Look-up*: for each stored answer $g_i\sigma$ current and FUTURE compute resolvent using the fact $g_i\sigma \leftarrow$
- else create a new table entry with call g_i and with empty list of answers compute resolvents with program clauses i.e. with clause $h \leftarrow B$, derive $B\sigma, \square_i, G\sigma$ notice special marker \square_i inserted
- if g_i is \square_j then a subrefutation for some call g_j is completed applied sequence of substitutions $\sigma_j, \dots, \sigma_k$ $g_j\sigma_j \dots \sigma_k$ is answer for table entry for g_j

avoids non-termination due to loops

always terminates for (Datalog) programs:

least Herbrand model as semantics

XSB-Prolog: declare predicates for tabling

$a(X, Y) \leftarrow p(X, Y)$
 $a(X, Y) \leftarrow a(X, Z), p(Z, Y)$
 $p(a, b) \leftarrow$
 $p(b, c) \leftarrow$
 $p(c, a) \leftarrow$

$\leftarrow a(a, A)$			
entry 0 $a(a, A)$			
$\{X_1/a, Y_1/A\}$	$\{X_2/a, Y_2/A\}$		
$\leftarrow p(a, A), \square_0$	$\leftarrow a(a, Z_2), p(Z_2, A), \square_0$		
entry 1 $p(a, A)$	Look-up 0		
$\{A/b\}$	$\{Z_2/b\}$	$\{Z_2/c\}$	$\{Z_2/a\}$
$\leftarrow \square_1, \square_0$	$\leftarrow p(b, A), \square_0$	$\leftarrow p(c, A), \square_0$	$\leftarrow p(a, A), \square_0$
1: answ $p(a, b)$	entry 2 $p(b, A)$	entry 3 $p(c, A)$	Look-up 1
$\leftarrow \square_0$	$\{A/c\}$	$\{A/a\}$	$\{A/b\}$
0: answ $a(a, b)$	$\leftarrow \square_2, \square_0$	$\leftarrow \square_3, \square_0$	$\leftarrow \square_0$
\square	2: answ $p(b, c)$	3: answ $p(c, a)$	0: answ $a(a, b)$
	$\leftarrow \square_0$	$\leftarrow \square_0$	exists
	0: answ $a(a, c)$	0: answ $a(a, a)$	\square
	\square	\square	

table entries:

entry 0: $a(a, A)$ answers: $a(a, b), a(a, c), a(a, a)$
 entry 1: $p(a, A)$ answers: $p(a, b)$
 entry 2: $p(b, A)$ answers: $p(b, c)$
 entry 3: $p(c, A)$ answers: $p(c, a)$

- OLD T: sound, complete
- OLD T-tree: shows “same” program behaviour as OLD-tree:
 - same set of selected atoms
 - for each of them: same computed answers
- one can use OLD T-tree to “collect” program behaviour instead of OLD-tree
- alternatively, one can consider OLD T-tree as abstraction of OLD-tree
- the concretisation function γ replaces look-up steps by refutation which was deriving the applied answers of the looked-up atom

take branch with Look-up of $a(a, A)$:

$\leftarrow a(a, A)$
 entry 0 $a(a, A)$
 $\{X_2/a, Y_2/A\}$
 $\leftarrow a(a, Z_2), p(Z_2, A), \square_0$
 Look-up 0
 \vdots

plug in the proof of $a(a, Z_2)$

$\leftarrow a(a, A)$ entry 0 $a(a, A)$ $\{X_2/a, Y_2/A\}$ $\leftarrow a(a, Z_2), p(Z_2, A), \square_0$	$\{X_3/a, Y_3/Z_2\}$ $\leftarrow p(a, Z_2), p(Z_2, A), \square_0$ $\{Z_2/b\}$ \vdots	$\{X_4/a, Y_4/Z_2\}$ $\leftarrow a(a, Z_4), p(Z_4, Z_2), p(Z_2, A), \square_0$ Look-up 0 $\{Z_4/b\} \{Z_4/c\} \{Z_4/a\}$ $\vdots \quad \vdots \quad \vdots$
---	---	---

plugged in part has another Look-up!

plugging in the whole tree is a never ending process

leads to infinite OLD-tree

An alternative formalisation of OLDT:

processing of $\leftarrow g_i, G$:

- If there is no tree for (renaming of) g_i , then create one.
- If the tree for g_i has an answer σ not yet used in this tree, then use it: resolve the selected atom with the (renamed) fact $g_i\sigma \leftarrow$.

1. $a(X, Y) \leftarrow p(X, Y)$
2. $a(X, Y) \leftarrow a(X, Z), p(Z, Y)$
3. $p(a, b) \leftarrow$
4. $p(b, c) \leftarrow$
5. $p(c, a) \leftarrow$

$\leftarrow a(a, A)$

(1) $\leftarrow p(a, A)$ (2) $\leftarrow a(a, Z_2), p(Z_2, A)$

(6) \square :

$a(a, b)$ (7)

(7) $\leftarrow p(b, A)$

(8) \square :

$a(a, c)$ (9)

(9) $\leftarrow p(c, A)$

(10) \square :

$a(a, a)$ (11)

(11) $\leftarrow p(a, A)$

(6) \square :

$a(a, b)$ (= 7)

$\leftarrow p(a, A)$

(3) \square :

$p(a, b)$ (6)

$\leftarrow p(b, A)$

(4) \square :

$p(b, c)$ (8)

$\leftarrow p(c, A)$

(5) \square :

$p(c, a)$ (10)

OLDT-trees can still be infinite
but are good basis for abstraction

α^{terms} : tuple of terms \rightarrow abstract tuples
extends to

α^{atoms} : set of atoms \rightarrow abstract atoms
extends to

abstraction of nodes in OLDT-tree

abstraction of table entries

extends to

abstraction of whole OLDT-tree

key operations:

- abstract resolution:
abstract goal \times clause \rightarrow abstract goal
- abstract look-up:
abstract goal \times abstract answer
 \rightarrow abstract goal
- creation of new table entry/new tree
- storage of a new answer

Example

abstraction of terms: ground terms abstracted as \mathcal{G}

see part 1, also for the abstract unification

$$a([], X, X) \leftarrow$$

$$a([E|U], V, [E|W]) \leftarrow a(U, V, W)$$

AOLDT-tree

$\leftarrow a(\mathcal{G}, \mathcal{G}, C)\}$ entry 1 $a(\mathcal{G}, \mathcal{G}, C)$ $\{\mathcal{G} = [], \mathcal{G} = X_1, C = X_1\}$ $\{X_1 = \mathcal{G}, C = \mathcal{G}\}$ $\leftarrow \square_1$ 1: answ $a(\mathcal{G}, \mathcal{G}, \mathcal{G})$ \square	$\{\mathcal{G} = [E_2 U_2], \mathcal{G} = V_2, C = [E_2 W_2]\}$ $\{E_2 = \mathcal{G}, U_2 = \mathcal{G}, V_2 = \mathcal{G}, C = [\mathcal{G} W_2]\}$ $\leftarrow a(\mathcal{G}, \mathcal{G}, W_2)\square_1$ Look-up 1 $\{\mathcal{G} = \mathcal{G}, \mathcal{G} = \mathcal{G}, W_2 = \mathcal{G}\}$ $\{W_2 = \mathcal{G}\}$ $\leftarrow \square_1$ 1: answ $a(\mathcal{G}, \mathcal{G}, \mathcal{G})$ exists \square
--	---

- can be infinite with this term abstraction
- by “expanding” Look-up:
infinite AOLD-tree is obtained
- every partial OLD-derivation can be mapped by α^{term} on a branch of this infinite AOLD-derivation

$$\begin{aligned}
&\leftarrow a([a, b], [c, d], C) \\
&\quad \leftarrow a(\mathcal{G}, \mathcal{G}, C) \\
&\{E_1 = a, U_1 = [b], V_1 = [c, d], C = [a|W_1]\} \\
&\quad \{E_1 = \mathcal{G}, U_1 = \mathcal{G}, V_1 = \mathcal{G}, C = [\mathcal{G}|W_1]\} \\
&\leftarrow a([b], [c, d], W_1) \\
&\quad \leftarrow a(\mathcal{G}, \mathcal{G}, W_1) \\
&\{E_2 = b, U_2 = [], V_2 = [c, d], W_1 = [b|W_2]\} \\
&\quad \{E_2 = \mathcal{G}, U_2 = \mathcal{G}, V_1 = \mathcal{G}, W_1 = [\mathcal{G}|W_2]\} \\
&\leftarrow a([], [c, d], W_2) \\
&\quad \leftarrow a(\mathcal{G}, \mathcal{G}, W_2) \\
&\{X_3 = [c, d], W_2 = [c, d]\} \\
&\quad \{X_3 = \mathcal{G}, W_2 = \mathcal{G}\}
\end{aligned}$$

□

□

Annoyances of AOLDT

- $\leftarrow a, A$ (a solved by Look-up)

$\leftarrow A_1 \dots \leftarrow A_n$

n calls to $\leftarrow A_i$

More efficient to do it once

for $\leftarrow \text{lub}(A_1, \dots, A_n)$

Possible but does not fit very elegantly

Note: if not all answers available: iterate until fixpoint

- $\leftarrow a, A$ (a solved by program clauses)

$\leftarrow B_1, A_1 \dots \leftarrow B_n, A_n$

...

$\leftarrow A_{1,1} \dots \dots \leftarrow A_{n,n_m}$

Different continuations $\leftarrow A_{i,j}$

Often preferable to merge them in one continuation

$\leftarrow \text{lub}(A_{1,1}, \dots)$ possible, but even less obvious

Relevance: e.g. in codegeneration: one does not want to jump to different versions on return from call to a

Generic systems use a different approach

Exercise

5. For the program:

$qs([], []) \leftarrow$.

$qs([X|L], S) \leftarrow \text{part}(X, L, L1, L2),$
 $\quad qs(L1, S1), qs(L2, S2),$
 $\quad \text{append}(S1, [X|S2], S).$

$\text{part}(X, [], [], []) \leftarrow$.

$\text{part}(X, [Y|L], [Y|L1], L2) \leftarrow X \geq Y, \text{part}(X, L, L1, L2).$

$\text{part}(X, [Y|L], L1, [Y|L2]) \leftarrow X < Y, \text{part}(X, L, L1, L2).$

$\text{append}([], L, L) \leftarrow$.

$\text{append}([X|U], V, [X|W]) \leftarrow \text{append}(U, V, W).$

compute the abstract OLDT tree using the \mathcal{G} abstraction for ground terms and for the call $qs(\mathcal{G}, S)$.

Give the set of abstract atoms being called during execution (abstract call-patterns for the query).

2.2.2 Abstracting LSLDT

LSLDT: Local SLD with Tabulation

Locality: environment in which clause is executed is restricted to the clause variables

consider $\leftarrow (a, A)\theta$

- for every clause $h \leftarrow B$ with $\sigma = mgu(h, a\theta)$
create local derivation $((h) \leftarrow B)\sigma$
 - implicit projection of σ on clause variables
 - eventually a cas τ is obtained
 - applying τ on h yields “lemma” $h\tau \leftarrow$
- resolution on $\leftarrow (a, A)\theta$ and lemma $h\tau \leftarrow$
gives $\leftarrow A\theta\mu$ with $\mu = mgu(a\theta, h\tau)$

add Tabulation:

- table entry: first step of local derivation:
 $((h) \leftarrow B)\sigma$
or, more convenient: the call $a\theta$
- answers: lemmas $h\tau$
- skip step 1 when table entry exists

Lifting to sets of substitutions $\Theta +$ abstraction:

ONE lemma as abstract answer (the *lub*) \rightsquigarrow
AND/OR graph

- OR-node $g_i(\bar{t})$
to its left: program point p_i with abstract state of clause variables: describing computation state when g_i is to be called
to its right: program point p_{i+1} , describing state of computation on successful completion of g_i
one child for each clause defining predicate of g_i
- AND-node: head of clause (child of OR-node), one child for each atom in body

Construction of AND-OR graph

Initialisation assume query is single atom q

create root of AND/OR graph: OR-node q adorned on the left with program point with abstraction of the set Θ of substitutions for which q is called, adorned on the right with \perp

Apply operations until fixpoint:

- Abstraction of built-in: operates on OR-node q_i with predicate of q_i a built-in
 - Let \overline{X} be the clause variables in p_i
 - Update abstraction for program point p_{i+1}
 - Correctness:
If $\theta \in \gamma(A_{p_i})$ and $q_i\theta$ has cas σ then $\theta\sigma|_{\overline{X}} \in \gamma(A_{p_{i+1}})$
If $\leftarrow (q_i, q_{i+1}, \dots)\theta$ is a state in a local derivation,
then $\leftarrow (q_{i+1}, \dots)\theta\sigma$ is the next state

- Create table entry: operates on OR-node q_i for defined predicate for which $\langle q_i, A_{p_i} |_{Var(q_i)} \rangle$ is not in Table
 - Create table entry for $\langle q_i, A_{p_i} |_{Var(q_i)} \rangle$,
 - Answer is initialised as $\langle q_i, \perp \rangle$
 - If q_i is leaf, extend the graph with structure for each clause $h_j \leftarrow b_{1,j}, \dots, b_{k,j}$:
AND-node h_j is child of q_i
children of h_j : OR-nodes $b_{1,j}, \dots, b_{k,j}$
 - Abstractions in new program points: initialised as \perp

- Procedure Entry: operates on OR-node q_i which has children
 - For each clause $h_j \leftarrow b_{1,j}, \dots, b_{k,j}$ with variables \bar{Y}
 - Abstraction $A_{p_{1,j}}$ is updated (initialisation of local derivation)
 - Correctness:
If $\theta \in \gamma(A_{p_i})$ and $\sigma = mgu(q_i\theta, h_j)$ then $\sigma|_{\bar{Y}} \in \gamma(A_{p_{1,j}})$
If $\leftarrow (q_i, q_{i+1}, \dots)\theta$ is a state in a local derivation, then
there is another local derivation starting with $\leftarrow (b_{1,j}, \dots, b_{k,j})\sigma$

- Procedure Exit: operates on OR-node q_i which has children
 - Update answer for table entry $\langle q_i, A_{p_i} | Var(q_i) \rangle$
 - Correctness: for the clauses $h_j \leftarrow b_{1,j}, \dots, b_{k,j}$ If $\theta \in \gamma(A_{p_i})$ and $\tau \in \gamma(A_{p_{k,j+1}})$ and $\sigma = mgu(q_i\theta, h_j\tau)$ then $q_i\theta\sigma \in \gamma(\langle q_i, A_{out} \rangle)$ with $\langle q_i, A_{out} \rangle$ the stored answer
If $\leftarrow (q_i, q_{i+1}, \dots)\theta$ is a state in a local derivation and the local derivation for the j^{th} clause ends with cas τ , then $q_i\theta\sigma = h_j\tau$ is a lemma
- Look-up: operates on OR-node q_i for which table entry $\langle q_i, A_{p_i} | Var(q_i) \rangle$ exists with answer $\langle q_i, A_{out} \rangle$
 - Updates $A_{p_{i+1}}$
 - Correctness:
If $\theta \in \gamma(A_{p_i})$ and $q_i\theta\tau \in \gamma(\langle q_i, A_{out} \rangle)$ then $\theta\tau \in \gamma(A_{p_{i+1}})$
If $\leftarrow (q_i, q_{i+1}, \dots)\theta$ is a state in a local derivation and $q_i\theta\tau$ is a lemma then $\leftarrow (q_{i+1}, \dots)\theta\tau$ is the next state

Fixpoint reached:

The state of a clause

$A_{p_1}, q_1, A_{p_2}, q_2, \dots, A_{p_k}, q_k, A_{p_{k+1}}$ describes all local derivations starting with

$(h) \leftarrow (q_1, q_2, \dots, q_k)\theta_1$ with $\theta_1 \in \gamma(A_{p_1})$

Indeed, correctness condition ensures:

- $((h) \leftarrow q_2, \dots, q_k)\theta_2$ with $\theta_2 \in \gamma(A_{p_2})$
- \vdots
- $((h) \leftarrow q_k)\theta_k$ with $\theta_k \in \gamma(A_{p_k})$
- $((h) \leftarrow \theta_{k+1})$ with $\theta_{k+1} \in \gamma(A_{p_{k+1}})$

$h\theta_{k+1}$ is a lemma to be used for solving the call to this clause

efficiently finding fixpoint requires synchronisation between different operations:

- Most important: avoid as long as possible to use answer lemma
Indeed, if used and afterwards updated then work has to be redone
However, redoing work (= fixpoint iteration) cannot always be avoided, especially in case of recursion
- Basic strategy: the prolog-execution strategy
But parallel evaluation of alternative definitions instead of backtracking
If call which is not yet tabled: procedure entry: analyse the defined clauses, non-recursive ones first
Wait with procedure exit until all definitions analysed
Wait with look-up until procedure-exit
- If blocked due to recursion then do
 - procedure exit
 - look-up of blocked calls
 - procedure-exit and if update of answer then redo blocked calls

Iterate until fixpoint

Systems differ in work to be redone when an already used lemma (say about q) is updated

Simple: reanalyse all definitions for all predicates p calling q

Better: only reanalyse the clauses of p calling q

Best: start reanalysis with look-up from program point preceding the call to q

But more bookkeeping (keeping the whole AND-OR structure is never done, too large)

Trade off between amount of bookkeeping and work of recomputing

What is best depends on the used abstraction

1. $dl(X, L, R) \leftarrow L = [X|R]$
2. $dl(X, L, R) \leftarrow L = [Y|Ls], R = [Y|Rs], dl(X, Ls, Rs)$
3. $pm(L, P) \leftarrow L = [], P = []$
4. $pm(L, P) \leftarrow P = [X|Ps], dl(X, L, R), pm(R, Ps)$

toplevel:

$\leftarrow pm(A, B); \{A = \mathcal{G}\}$

using (5): $\square; \{A = \mathcal{G}, B = \mathcal{G}\}$

Derivation for clause (3) with $\{L = \mathcal{G}\}$

$pm(L, P) \leftarrow L = [], P = []; \{L = \mathcal{G}\}$

$pm(L, P) \leftarrow P = []; \{L = \mathcal{G}\}$

$pm(L, P) \leftarrow; \{L = \mathcal{G}, P = \mathcal{G}\}$ i.e. Lemma (5): $pm(\mathcal{G}, \mathcal{G}) \leftarrow$

derivation for clause (4) with $\{L = \mathcal{G}\}$

$pm(L, P) \leftarrow P = [X|Ps], dl(X, L, R), pm(R, Ps); \{L = \mathcal{G}\}$

$pm(L, P) \leftarrow d(X, L, R), pm(R, Ps); \{L = \mathcal{G}, P = [X|Ps]\}$

Using (6):

$pm(L, P) \leftarrow pm(R, Ps); \{L = \mathcal{G}, P = [\mathcal{G}|Ps], X = \mathcal{G}, R = \mathcal{G}\}$

Using (5):

$p(L, P) \leftarrow; \{L = \mathcal{G}, X = \mathcal{G}, R = \mathcal{G}, P = \mathcal{G}, Ps = \mathcal{G}\}$

same lemma as (5)

derivation for clause (1) with $\{L = \mathcal{G}\}$

$d(X, L, R) \leftarrow L = [X|R]; \{L = \mathcal{G}\}$

$d(X, L, R) \leftarrow; \{L = \mathcal{G}, X = \mathcal{G}, R = \mathcal{G}\}$ i.e. Lemma (6): $d(\mathcal{G}, \mathcal{G}, \mathcal{G})$

derivation for clause (2) with $\{L = \mathcal{G}\}$

$d(X, L, R) \leftarrow L = [Y|Ls], R = [Y|Rs], d(X, Ls, Rs); \{L = \mathcal{G}\}$

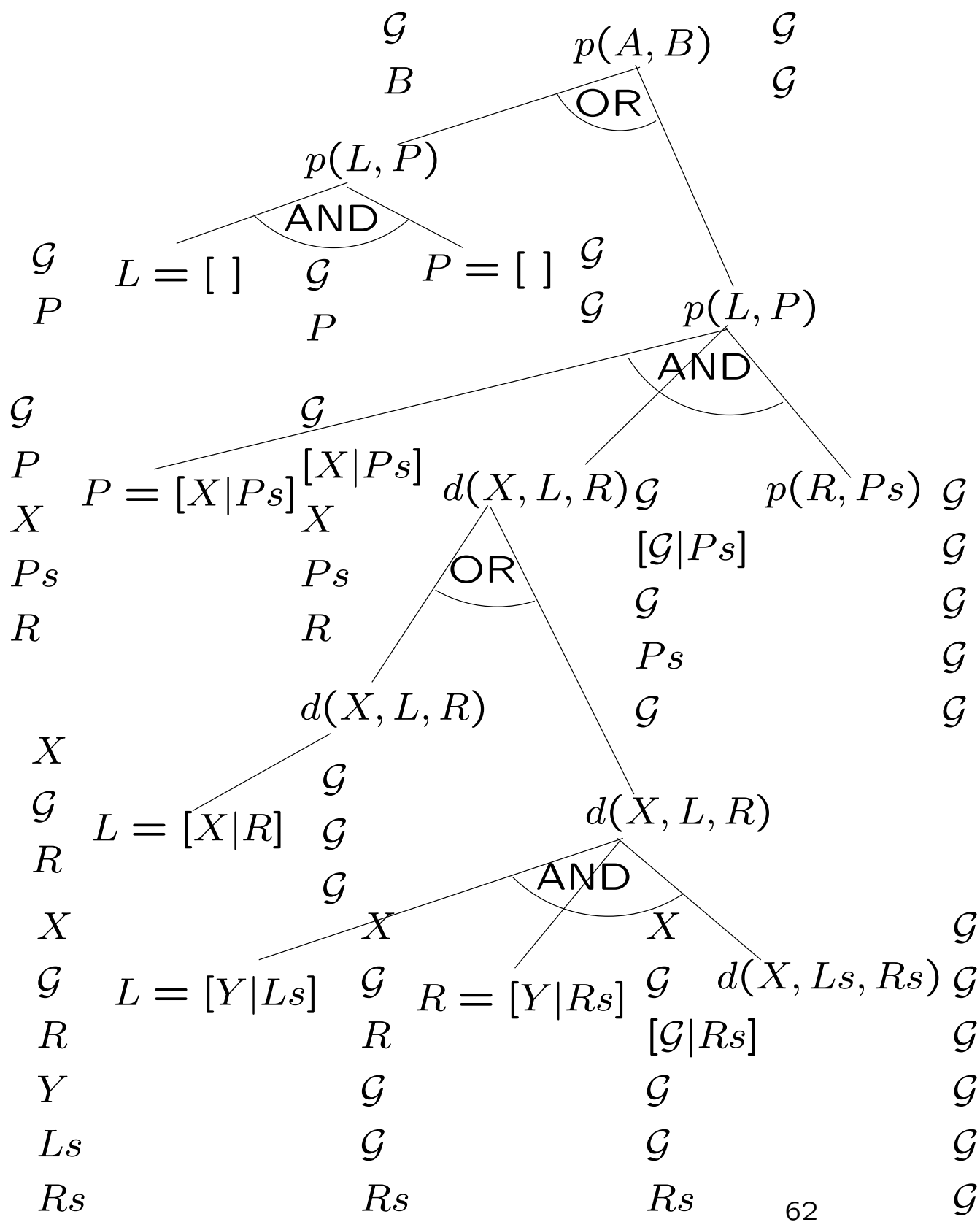
$d(X, L, R) \leftarrow R = [Y|Rs], d(X, Ls, Rs); \{L = \mathcal{G}, Y = \mathcal{G}, Ls = \mathcal{G}\}$

$d(X, L, R) \leftarrow d(X, Ls, Rs); \{L = \mathcal{G}, Y = \mathcal{G}, Ls = \mathcal{G}, R = [\mathcal{G}|Rs]\}$

Using (6)

$d(X, L, R) \leftarrow; \{L = \mathcal{G}, Y = \mathcal{G}, Ls = \mathcal{G}, R = \mathcal{G}, X = \mathcal{G}, Rs = \mathcal{G}\}$

same lemma as (6)



Why “graph”: nodes which are solved by look-up refer in fact to nodes where the call is solved

Clause with abstractions in program points is unit of info for compiler

Graph has different instances of same clause when clause is called with different abstractions

Leads to different versions (renaming of predicates)

Better not different versions for recursive call avoid by “widening” on tabled entry (lub or more aggressive)

Number of updates of answer lemma can also be limited through widening

Two basic operations:

- Projection:
 - Abstraction $(A) \times$ set of variables $(\bar{Y}) \rightarrow$ abstraction $(A|_{\bar{Y}})$
 - Reduces the set of variables described by abstraction
usually rather trivial operation
 - Correctness: Let \bar{X} be a set of program variables abstracted by A and \bar{Y} a subset
If $\bar{X}\theta \in \gamma(A)$ then $\bar{Y}\theta \in \gamma(A|_{\bar{Y}})$
- Unification:
 - Abstraction $(A_{in}) \times$ equation $(s = t) \rightarrow$ abstraction (A_{out})
 - Equation over variables described by abstraction
 - Correctness
If $\bar{X}\theta \in \gamma(A_{in})$ and $mgu(s\theta = t\theta) = \sigma$
then $\bar{X}\theta\sigma \in \gamma(A_{out})$

- procedure entry
 - Table entry computed by projection
 - Unification $call = head$ where head variables are initialised with abstraction of “free” variable
 - Freeness can be exploited, even if not expressed by abstraction
in particular, with arguments of call and head variables: a simple renaming
 - Procedure entry: typical for topdown
- Lemma generation (top down and bottom up)

If terms t_1, \dots, t_k in head:
then unification $X_1 = t_1, \dots, X_k = t_k$ with X_1, \dots, X_k new (free) variables followed by projection on X_1, \dots, X_k
else projection on head variables

- Look up (top down and bottom up)
 - Lemma $q(Y_1, \dots, Y_k)$ with abstraction A_l over Y_1, \dots, Y_k
 - Abstraction A over X_1, \dots, X_n ,
 - Call $q(X_{i1}, \dots, X_{ik})$
 - Extend A with A_l
 - Unification $X_{i1} = Y_1, \dots, X_{ik} = Y_k$
 - Projection on X_1, \dots, X_n

Exploit freeness of the X_{ij} if present:
(e.g. first occurrence)

Process these equations first:

project out X_{ik} , rename Y_k into X_{ik}

Exercise

6. For the program:

$qs([], []) \leftarrow$.

$qs([X|L], S) \leftarrow$ part($X, L, L1, L2$),
 $qs(L1, S1), qs(L2, S2)$,
 append($S1, [X|S2], S$).

$part(X, [], [], []) \leftarrow$.

$part(X, [Y|L], [Y|L1], L2) \leftarrow X \geq Y, part(X, L, L1, L2)$.

$part(X, [Y|L], L1, [Y|L2]) \leftarrow X < Y, part(X, L, L1, L2)$.

$append([], L, L) \leftarrow$.

$append([X|U], V, [X|W]) \leftarrow append(U, V, W)$.

compute the abstract LSLDT tree using the \mathcal{G} abstraction for ground terms and for the call $qs(\mathcal{G}, S)$.

Give the set of abstract atoms being called during execution (abstract call-patterns for the query).

7. For the same program, use the automated approach to prove left-termination of the calls abstracted by $qs(\mathcal{G}, S)$.