Abstract Interpretation

Abstract Interpretation

- Simulate computation (of the program) on a non-standard domain

Casting out of nine

- Detect errors in calculation of arithmetic expression (+, - ,*)
  - Calculate the remainder of sum of digits in an operand divided by 9
  - Take module 9 of each intermediate and final results
  - The result obtained as above agree with that of *normal calculation* module 9.

Casting out of nine

Concrete semantics

- Domain: Z
- $\boldsymbol{n} \rightarrow N(\boldsymbol{n})$
- **+** $\rightarrow$ *add*
- **-** $\rightarrow$ *sub*
- ***** $\rightarrow$ *mult*

Abstract semantics

- Domain: $Z_9 = \{0,\ldots,8\}$
- $n \rightarrow N(n) \bmod 9$
- + $\rightarrow$ add_9
- **-** $\rightarrow$ sub_9
- * $\rightarrow$ $mult_9$

Casting out of nine

$X \text{ add}_9 \; Y = (X \text{ add } Y) \bmod 9$

$X \text{ sub}_9 \; Y = (X \text{ sub } Y) \bmod 9$

$X \text{ mult}_9 \; Y = (X \text{ mult } Y) \bmod 9$

$\alpha : Z \rightarrow Z_9$

$\alpha(0) = 0$

$\alpha(8) = 8$

…

$\alpha(9) = 0$

$\alpha(10a \pm b) = \alpha(a \pm b)$

Casting out of nine

Concrete semantics

$[\ ] : \text{Exp} \rightarrow Z$

$[\boldsymbol{n}] = N(\boldsymbol{n})$

$[\ e_1+e_2] = [e_1] \text{ add } [e_2]$

$[\ e_1-e_2] = [e_1] \text{ sub } [e_2]$

$[\ e_1*e_2] = [e_1] \text{ mult } [e_2]$

Abstract semantics

$[\ ]_9 : \text{Exp} \rightarrow Z_9$

$[\boldsymbol{n}]_9 = \alpha(N(\boldsymbol{n}))$

$[\ e_1+e_2]_9=[e_1]_9 \text{ add}_9 [e_2]_9$

$[\ e_1-e_2]_9=[e_1]_9 \text{ sub}_9 [e_2]_9$

$[\ e_1*e_2]_9= [e_1]_9 \text{ mult}_9 [e_2]_9$

Exact Abstraction

$$\begin{tikzcd}[column sep=huge, row sep=huge]
\boxed{Z} \arrow[r, "{[\ ]}"] \arrow[d, "\alpha"] \arrow[dr, phantom, "="] & \boxed{Z} \arrow[d, "\alpha"] \\
\boxed{Z_9} \arrow[r, "{[\ ]_9}"'] & \boxed{Z_9}
\end{tikzcd}$$

Abstract Interpretation

Key points so far:

- There are two semantics
  - Concrete semantics models normal execution;
  - Abstract semantics gives analysis;
    - Constants in concrete domain is abstracted by constants in abstract domain
    - An n-ary operation on concrete domain is abstracted by an n-ary operation on abstract domain
  - Correctness is established with respect to a relationship between the abstract and the concrete domains.

Parity Analysis
(Jones & Nielson)

A: **while** n ≠ 1 **do**

B: **if** n is even

then (C: n := n÷2; D:)

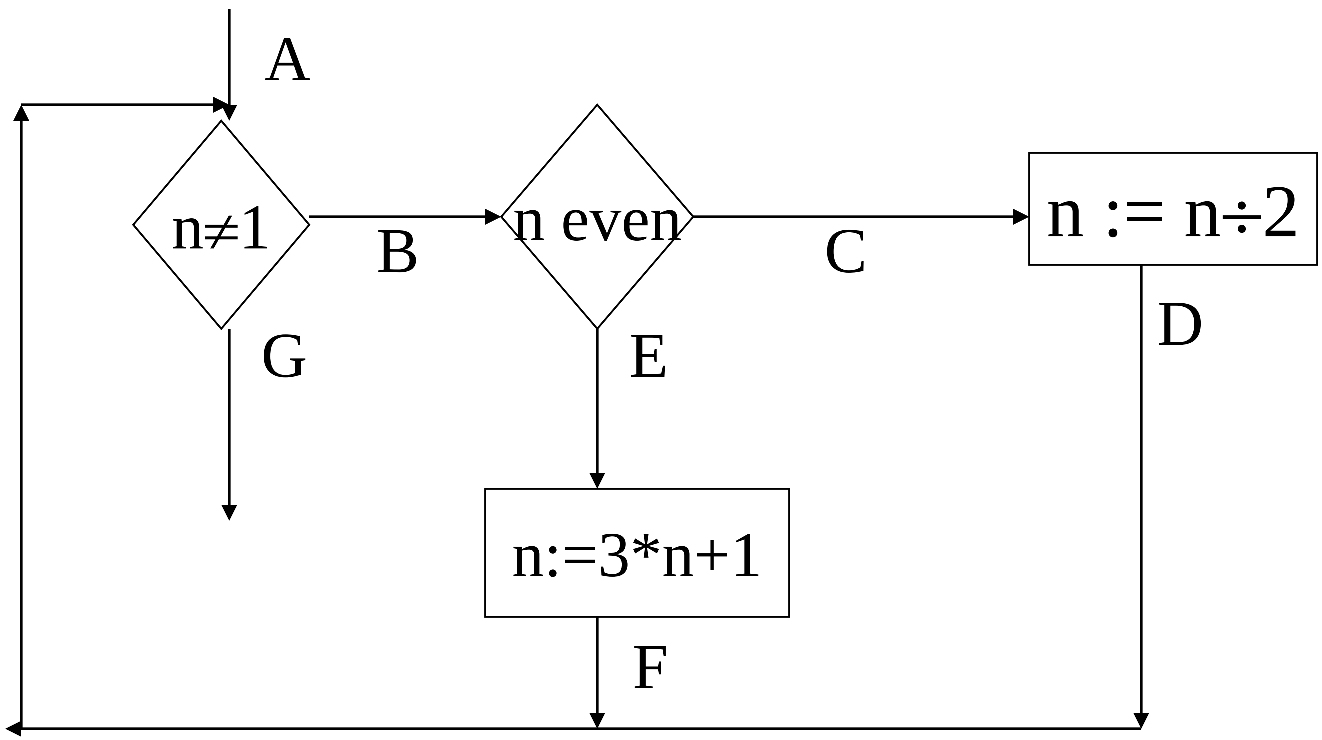
else (E: n := 3*n+1; F:)

fi

od

G:

Flowchart



Operational semantics

- A ***state*** is a pair of program point and a ***store***
- Each elementary block (test, assignment) defines a transition
  – $\text{next}_{n \neq 1}(\langle A, \{n \rightarrow 5\}\rangle) = \langle B, \{n \rightarrow 5\}\rangle$
  – $\text{next}_{n \neq 1}(\langle A, \{n \rightarrow 1\}\rangle) = \langle G, \{n \rightarrow 1\}\rangle$

Collecting semantics

Each program point is associated with the set of stores that may be obtained during execution of the program

$acc_p = \{s \mid next^*(<p_0,s_0>) = <p,s> \}$

Collecting semantics

Domain: $(2^N, \subseteq, \cup, \cap, N, \emptyset)$

$acc_A = S_0$
$acc_B = (acc_A \cup ace_D \cup acc_F) \cap (N \backslash \{1\})$
$acc_C = acc_B \cap \{2m \mid m \in N\}$
$acc_D = f_{n \div 2}(acc_C)$
$acc_E = acc_B \cap \{2m+1 \mid m \in N\}$
$acc_F = f_{3n+1}(acc_E)$
$acc_G = (acc_A \cup ace_D \cup acc_F) \cap \{1\}$

$f_{n \div 2}(S) = \{n \div 2 \mid n \in S\}$
$f_{3n+1}(S) = \{3n+1 \mid n \in S\}$

Abstract semantics

Domain: $<P,\leq,\vee,\wedge,\Delta,\nabla)$ where

$P = \{odd, even, \Delta, \nabla\}$

$\Delta \leq \Delta$, $\Delta \leq$ odd, $\Delta \leq$ even, $\Delta \leq \nabla$,

odd $\leq$ odd, even $\leq$ even

odd $\leq \nabla$, even $\leq \nabla$, $\nabla \leq \nabla$

Abstraction

$$\alpha(S) = \begin{cases} \Delta, & \text{if } S = \{\} \\ \text{odd}, & \text{if } S \subseteq \{1,3,\ldots\} \\ \text{even}, & \text{if } S \subseteq \{0,2,\ldots\} \\ \nabla, & \text{else} \end{cases}$$

Abstract semantics

$abs_A = \alpha(S_0)$

$abs_B = (abs_A \vee abs_D \vee abs_F) \wedge \alpha(N\backslash\{1\})$

$abs_C = abs_B \wedge \alpha(\{2m \mid m \in N\})$

$abs_D = \underline{f}_{\underline{n \div 2}} (abs_C)$

$abs_E = abs_B \wedge \alpha(\{2m+1 \mid m \in N\})$

$abs_F = \underline{f}_{\underline{3n+1}}(abs_E)$

$abs_G = (abs_A \vee abs_D \vee abs_F) \wedge \alpha\{1\}$

Abstract operations

$f_{n÷2}$ (abs) = if abs=Δ then Δ else ∇

f_{3n+1}(abs) = Δ, if abs= Δ

∇, if abs= ∇

even, if abs=odd

odd, if abs=even

Abstract semantics

$abs_A = \alpha(S_0)$

$abs_B = (abs_A \vee abs_D \vee abs_F) \wedge \nabla$

$abs_C = abs_B \wedge \text{even}$

$abs_D = \underline{f}_{n \div 2}\,(abs_C)$

$abs_E = abs_B \wedge \text{odd}$

$abs_F = \underline{f}_{3n+1}(abs_E)$

$abs_G = (abs_A \vee abs_D \vee abs_F) \wedge \text{odd}$

Safety

- α is not homomorphic, so we could not have exact abstraction as in casting out of nine!
- Global safety: $\alpha(\text{ lfp } F_{acc}) \leq \text{lfp } F_{abs}$
- Local safety: each abstract operation safely abstracts corresponding concrete operation.
  - $\alpha(S_1 \cup S_2) \leq \alpha(S_1) \vee \alpha(S_2)$

Galois connection

Instead of an abstraction function α, another way to relate abstract and concrete domains is via a concretization function γ from the abstract domain to the concrete domain.

$$
\gamma(abs) = \begin{cases} \{\}, & \text{if } abs = \Delta, \\ \{2m+1 \mid m \in N\}, & \text{if } abs = odd, \\ \{2m \mid m \in N\}, & \text{if } abs = even, \\ N, & \text{if } abs = \nabla \end{cases}
$$

Galois connection

Functions α and γ satisfy:

$$S \subseteq \gamma(\alpha(S))$$

$$\alpha(\gamma(abs)) \leq abs$$

This is called a Galois connection

Galois Connection

Let $\langle C, \leq_C \rangle$ and $\langle A, \leq_A \rangle$ be posets. The pair $\alpha: C \rightarrow A$ and $\gamma: A \rightarrow C$ is a Galois *connection* iff

$$c \leq_C \gamma(\alpha(c)) \text{ for each } c \text{ in } C$$

$$\alpha(\gamma(a)) \leq_A a \text{ for each } a \text{ in } A$$

If $\alpha(\gamma(a)) = a$ then (α,γ) is a Galois *insertion*.

Parity Analysis Revisited

(α,γ) is a Galois insertion:

Let S be a set of natural numbers. Then It can be easily verified that $S \subseteq \gamma(\alpha(S))$.

Let P be any parity. Then $P = \alpha(\gamma(P))$ follows by simple case analysis.

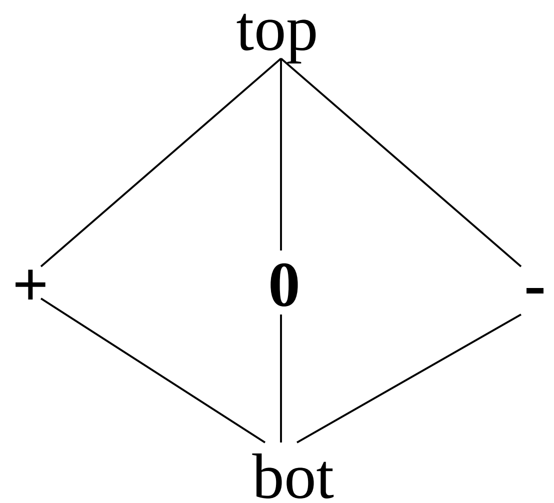
Alternative Definition

- Let $<C, \leq_C>$ and $<A, \leq_A>$ be posets. The pair α: $C \rightarrow A$ and γ: $A \rightarrow C$ is a Galois *connection* iff

$$\alpha(c) \leq a \Leftrightarrow c \leq \gamma(a)$$

Galois Connection for Sign Analysis

- Let Sign = { top, bot, **+**, **0**, **-** }



Values = $(2^Z, \subseteq, \cup, \cap, N, \varnothing)$

$$\alpha_{sign}(X) = \begin{cases} \text{bot}, & \text{if } X = \varnothing \\ +, & \text{if } X \subseteq \{1,2,\ldots\} \\ 0, & \text{if } X = \{0\} \\ -, & \text{if } X \subseteq \{\ldots, -2,-1\} \\ \text{top}, & \text{else} \end{cases}$$

$$\gamma_{sign}(S) = \begin{cases} \varnothing, & \text{if } S = \text{bot} \\ \{1,2\ldots\} & \text{if } S = + \\ \{0\} & \text{if } S = 0 \\ \{\ldots,-2,-1\} & \text{if } S = - \\ Z & \text{else} \end{cases}$$

$(\alpha_{sign,}\gamma_{sign})$ is a Galois connection (insertion).

Properties of Galois connections

- $\alpha\gamma\alpha = \alpha$
- $\gamma\alpha\gamma = \gamma$
- $\alpha\gamma$ is a lower closure.
- $\gamma\alpha$ is an upper closure.
- If $<A, \leq_A>$ is a complete lattice then $\gamma^*(A)$ is a Moore family.
- α and γ determines each other:

  $\gamma(a) = \vee_C\{ c \mid \alpha(c) \leq_A a \}$

  $\alpha(c) = \wedge_A\{ a \mid c \leq_C \gamma(a) \}$

- α preserves l.u.b.
- γ preserves g.l.b.
- α is onto iff γ is one-to-one (Galois insertion)
- Every Galois connection can be ***reduced*** into a Galois insertion.
  - Independent attribute sign analysis.

Operational semantics - revisited

- Each elementary block is a transition function.
  - A state is a pair of a program point and a store. s=<p,m> where p is a point and m is a memory store.
  - The block maps a state to another state:
    - [x:=e] next(<p,m>)=<q,m[x→***A***[e]m]>
    - [skip] next(<p,m>)=<q,m>
    - [b] next(<p,m>)=<q_{true},m> if ***B***[b]m=tt

      next(<p,m>)=<q_{false},m> if ***B***[b]m=ff

    Note that p labels the entry of the block while q with/without subscript labels the entry of the block to be executed next.

Properties of memory stores

- Many program analyses derive invariants of memory stores for each program point.
- An invariant is
  - a ***logic formula*** that is satisfied by all memory stores that may be obtained;
  - a set of memory stores that includes all memory stores that may be obtained
- These two views are both useful.

Collecting semantics

Lifting next relation

- [x:=e]

  $Next(<p,M>)=<q, \{m[x \rightarrow \boldsymbol{A}[e]m] \mid m \in M\}>$

- [skip]

  $Next(<p,M>)=<q,M>$

- [b]

  $Next(<p,M>) = <q_{true}, M \cap \{m \mid \boldsymbol{B}[b]m=tt\}>$

  $Next(<p,M>) = <q_{false}, M \cap \{m \mid \boldsymbol{B}[b]m=ff\}>$

Collecting semantics

Collecting all states with the same points together:

$$X_q = \cup\{X \mid Next(p,X_p) = (q,X) \text{ and } p \rightarrow q\}$$

Where X_q is a set of memory stores associated with q and $p \rightarrow q$ indicates that the block labeled q may be executed immediately after that labeled p.

Abstract semantics

Logical view

- [x:=e]

  next(<p,f>)=<q, sp(x:=e,f)>

- [skip]

  next(<p,f>)=<q,f>

- [b]

  next(<p,f>) = <q_{true}, $f \wedge b$>

  next(<p,f>) = <q_{false}, $f \wedge \neg b$>

Abstract semantics

Group all formulae associated with the same program point:

$$f_q = \vee \{ X \mid \mathbf{next}(p, X_p) = (q, X) \text{ and } p \rightarrow q \}$$

Abstract semantics

- Logic formulae without restrict are too general. Analysis requires restriction on logic formulae. This restriction in turn requires operations next, sp, logical connectives to be approximated.
- In a sign analysis, formulae are formed of primitive formulae (x is positive), (x is negative) together with logical connectives.
- Logic formulae are then represented in an isomorphic domain (abstract domain).

Sign analysis

```
while ① (x>0)
do
② x := x-1;
③ z := y*y
od;
④
z := -z
⑤
```

A logic formula might be x is positive, y negative and z positive.

This can be represented as an abstract state:

$\{x \rightarrow +, y \rightarrow -, z \rightarrow +\}$

Global conditions

- Both concrete and abstract semantics are least fixed points.

  $\alpha(\text{lfp } f_C) \leq_A \text{lfp } f_A$

  $\text{lfp } f_C \leq_C \gamma(\text{lfp } f_A)$

  A sufficient (global) condition is $f_C \leq \gamma\, f_A\, \alpha$ or equivalently $\gamma\, f_C\, \alpha \leq f_A$.

Local conditions

- Often f_C and f_A are defined in the same way: an operation o_C is applied whenever o_A is applied.
- Global safety condition can be reduced into a set of local safety conditions.

Best abstract operations/analysis

- Global -- $f_A = \gamma\, f_C\, \alpha$
- Local -- $o_A = \gamma_0\, o_C \langle\alpha_1, \ldots, \alpha_n\rangle$
- Best analysis maybe un-computable.

Independent attribute versus relational analysis

- Independent attribute – relationship between variables are not captured.
- Relational – keep track of relationship between variables using a power domain construction.
- Example

  if rand() then (x:=1;y:=1) else (x:=-1;y:=-1)

Disjunctive completion

- Analysis precision can be improved by using a more rich domain.
- Disjunctive completion of an abstract domain allows disjunction in the concrete domain to be precisely represented as disjunction in the abstract domain.
- Example - Sign analysis

Martelli-Montanari unification

Apply following steps until no change

- $X=X::E \rightarrow E$
- $X=t::E \rightarrow$

  if occurs(X,t) then failure else $X=t::E[X/t]$
- $f(t_1,\ldots,t_n)=X::E \rightarrow X=f(t_1,\ldots,t_n)::E$
- $f(t_1,\ldots t_n)=g(s_1,\ldots s_m)::E \rightarrow$

  if $f/n = g/m$ then $t_1=s_1,\ldots t_n=s_n::E$ else failure

Basic operation in LP

- Unifying an atom $\theta(A)$ with another atom H where H and $\theta(A)$ do not share variable.
- Equivalently, unifying $A=H::\theta$. In general, unifying $E \cup \theta$

Simple Groundness Analysis

- Derive which program variables are definitely ground. Abstract substitution is
  - a mapping from the set VI of variable of interest to domain {G,top} with a bot to indicate failure or
  - a subset of VI with a bot to indicate failure
  - What is Galois connection?

Abstract unification

- Computes an output abstract substitution from a set of equations and an input abstract substitution.
- Abstract terms: terms with ***G*** as a constant

Apply following steps until no change

- $X=X::E \rightarrow E$
- $X=t::E \rightarrow$

  if occurs(X,t) then failure

  else if t is ground then $X=\boldsymbol{G}::E$

  else $X=t::E[X/t]$
- $f(t_1,\ldots,t_n)=X::E \rightarrow X=f(t_1,\ldots,t_n)::E$
- $f(t_1,\ldots t_n)=g(s_1,\ldots s_m)::E \rightarrow$

  if $f/n = g/m$ then $t_1=s_1,\ldots t_n=s_n::E$ else failure
- $\boldsymbol{G}=g(s_1,\ldots s_m)::E \rightarrow s_1=\boldsymbol{G},\ldots s_n = \boldsymbol{G}::E$
- $g(s_1,\ldots s_m)=\boldsymbol{G}::E \rightarrow s=\boldsymbol{G},\ldots s_n=\boldsymbol{G}::E$

Retain only equations of the form $Y=\boldsymbol{G}$.

Example

- $X=f(a,U)$
- $Y=f(V,b)$
- $X=Y$
- $g(U,W)=g(V,h(Z))$

Example

Given the append program

append([ ], X, X).

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

Construct an SLD tree for the abstract query $\leftarrow$ append(***G***,***G***,V) using abstract unification instead of concrete unification. Which abstract answers do you get?

SOS

$$(\sigma,(A,G)) \xrightarrow{LD} (\theta,(B,mk(A,\sigma,H),G)) \quad if \quad \begin{array}{c} (H \leftarrow B) \in P \\ \theta = mgu(A\sigma\rho,H) \neq fail \end{array}$$

$$(\vartheta,(mk(A,\sigma,H),G)) \xrightarrow{LD} (\sigma \quad mgu(A\sigma,H\vartheta\rho),G)$$

Observation

- A goal in the alternative SOS is of the form

$$\mathbf{(\theta_{n},B_{n}(A_{n-1},\theta_{n-1},H_{n})B_{n-1}\ (A_{n-2},\theta_{n-2},H_{n-1})B_{n-2}\ldots.(A_{1},\theta_{1},H_{2})B_{1})}$$

- The goal can be re-organized as

$$\mathbf{(\theta_{n},B_{n},H_{n})(\theta_{n-1},A_{n-1},B_{n-1},H_{n-1})\ldots(\theta_{1},A_{1},B_{1},\blacklozenge))}$$

  where each B_i can be □ and is a suffix of a clause whose head is H_i. ♦ is falsity and is the head of the query.

SOS

$$(\sigma,(A,B'),H')S \longrightarrow (\theta,B,H)(\sigma,(A,B'),H')S \quad \text{if} \quad \begin{array}{c}(H \leftarrow B) \in P \\ \theta = mgu(A\sigma\rho, H) \neq fail\end{array}$$

$$(\vartheta,\langle\ \rangle,H)(\sigma,(A,B'),H')S \longrightarrow (\sigma \quad mgu(A\sigma, H\vartheta\rho), B', H')S$$

Example

append([ ], X, X).

append([X|U],V,[X|W]) ← append(U,V,W).

← append([1,2],[a,b,c],L).

Encapsulating procedure-entry and procedure-exit operations

$$unify(a, \sigma, b, \theta) = \theta \circ mgu(a\sigma\rho, b\theta)$$

SOS

$$(\sigma,(A,B'),H')S \longrightarrow (\theta,B,H)(\sigma,(A,B'),H')S \quad if \quad \begin{array}{c} (H \leftarrow B) \in P \\ \theta = unify(A,\sigma,H,\{\}) \neq fail \end{array}$$

$$(\vartheta,\langle\ \rangle,H)(\sigma,(A,B'),H')S \longrightarrow (unify(H,\vartheta,A,\sigma),B',H')S$$

Program flow graph

- Nodes are program points;
- Edge p $\rightarrow$ q indicates that q will be possibly visited immediately after p is visited.
- Call edges – models possible procedure-entry operation
- Return edges – models possible procedure-exit operation
- Let p be a program point in a clause. Then the suffix of the body to the right of p and the head of the clause are determined.

SOS

$$(\sigma, p)S \longrightarrow (\theta, q)(\sigma, p)S \quad if \quad \begin{array}{c} p \xrightarrow{call} q \\ \theta = unify(A(p), \sigma, H(q), \varepsilon) \neq fail \end{array}$$

$$(\sigma, p)(\omega, q-)S \longrightarrow (\theta, q)S \quad if \quad \begin{array}{c} p \xrightarrow{return} q \\ \theta = unify(H(p), \sigma, A(q-), \omega) \neq fail \end{array}$$

Accumulating substitutions

cunify(a,Θ,b,Ω) = {unify(a,θ,b,ω) | θ∈Θ∧ω∈Ω} \ {fail}

$$(\Sigma, p)S \longrightarrow (\Theta, q)(\Sigma, p)S \quad \mathit{if} \quad \begin{array}{c} p \xrightarrow{call} q \\ \Theta = \mathit{cunify}(A(p), \Sigma, H(q), \{\varepsilon\}) \end{array}$$

$$(\Sigma, p)(\Omega, q-)S \longrightarrow (\Theta, q)S \quad \mathit{if} \quad \begin{array}{c} p \xrightarrow{return} q \\ \Theta = \mathit{cunify}(H(p), \Sigma, A(q-), \Omega) \end{array}$$

Nilsson's Collecting semantics

Initial point: $X_\iota = \Theta_\iota$

For entry points:

$$X_q = \bigcup \{ cunify(A(p), X_p, H(q), \{\varepsilon\}) \mid p \xrightarrow{call} q \}$$

For other points:

$$X_q = \bigcup \{ cunify(H(p), X_p, A(q-), X_{q-}) \mid p \xrightarrow{return} q \}$$

Nilsson's Abstract semantics

- Concrete interpretation: $\left\langle 2^{Sub}, \subseteq, \varnothing, \bigcup, cunify, \{\varepsilon\} \right\rangle$
- Abstract interpretation: $\left\langle ASub, \leq, \perp, \vee, aunify, id \right\rangle$
- Galois connection: $\left\langle 2^{Sub}, \alpha, ASub, \gamma \right\rangle$

Nilsson's Abstract semantics

Initial point: $\mathbf{X}_\iota = \pi_\iota$

For entry points:

$$X_{\mathrm{q}} = \vee\{aunify(A(p), X_p, H(q), id) \mid p \xrightarrow{call} q\}$$

For other points:

$$X_{\mathrm{q}} = \vee\{aunify(H(p), X_p, A(q-), X_{q-}) \mid p \xrightarrow{return} q\}$$

Local Safety Conditions

- $\varepsilon \in \gamma(\mathrm{id})$

- $\mathrm{cunify}(a_1,\gamma(\pi_1),a_2, \gamma(\pi_2)) \subseteq \gamma(\mathrm{aunify}(a_1,\pi_1,a_2,\pi_2))$ for any atoms a1 and a2 and any abstract substitutions π_1 and π_2

Groundness Analysis

- A variable x is ground in a substitution σ iff $x\sigma$ contains no variable. Let vars(t) be the set of variables occurring in t. Then x is ground in σ iff $vars(x\sigma)=\{\}$
- Groundness analysis aims to identify which variables are ***definitely*** ground at a given program point.

Abstract domain

To describe a set of substitutions by

- a function from variables of interest to {**g**,**a**} ordered **g** $\leq$ **a** augmented with a bottom $\perp$
- Abstract domain ASub = (VI → {**g**,**a**})$_\perp$

$\perp \quad \leq \pi$ for any π in ASub

$\pi_1 \leq \pi_2$ iff $\pi_1(x) \leq \pi_2(x)$ for any x in VI

$\gamma(\perp) = \{\}$

$\gamma(\pi) = \{\theta \mid \forall x \in VI.(\pi(x)=\mathbf{g} \Rightarrow vars(x\theta)=\{\})\}$

Abstract domain

To describe a set of substitutions by

- a subset of the set variables of interest augmented with a bottom $\perp$
- Abstract domain ASub = $2^{VI}{}_{\perp}$

$\perp \leq \pi$ for any π in ASub

$\pi_1 \leq \pi_2$ iff $\pi_1 \supseteq \pi_2$

$\gamma(\perp) = \{\}$

$\gamma(\pi) = \{\theta \mid \forall x \in VI.(x \in \pi \Rightarrow vars(x\theta)=\{\})\}$

Abstract operations

- $id = \{\}$
- $aunify(a,\pi_1,b,\pi_2) =$

    let ψ be a tagging substitution ***in***

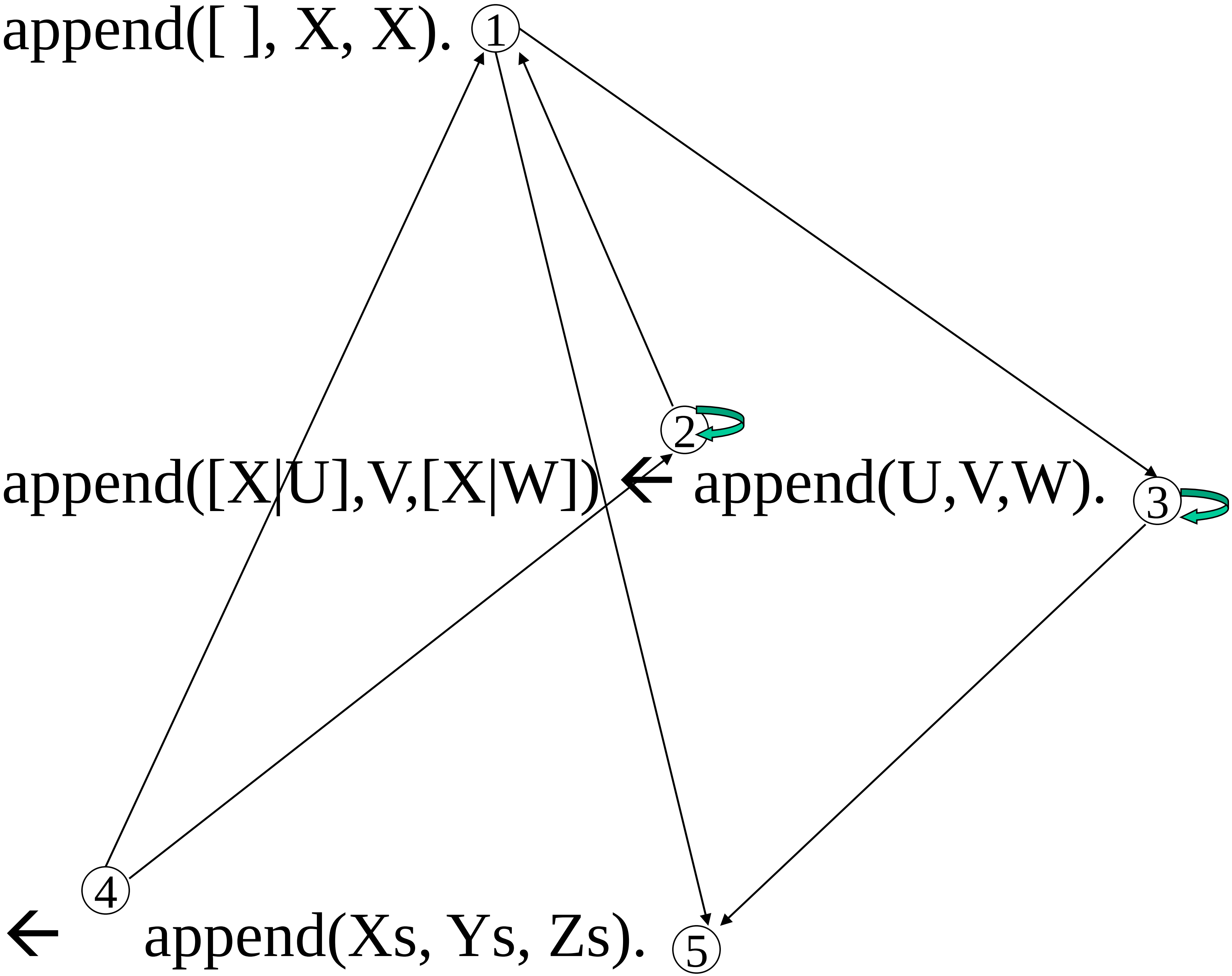
    let $E = mgu(a\psi,b)$ ***in***

    if E=fail then $\perp$

    else $up(E,down(E, \pi_1\psi\cup \pi_2)) \cap VI$

$up(E, \pi) = \pi \cup \{ x \mid ((x \rightarrow t)\in E) \text{ and } (vars(t)\subseteq\pi)\}$

$down(E, \pi) = \pi \cup \{ y \mid ((x\rightarrow t)\in E) \text{ and } (x\in\pi) \text{ and } (y\in vars(t))\}$


append([ ], X, X).
1
2
append([X|U],V,[X|W]) ← append(U,V,W).
3
4
← append(Xs, Ys, Zs).
5

a=append([],X,X)
b=append([X|U],V,X|W]
c=append(U,V,W)
d=append(Xs,Ys,Zs)

- X_1 = aunify(d,X_4,a,id) $\vee$ aunify(c,X_2,a,id)
- X_2 = aunify(d,X_4,b,id) $\vee$ aunify(c,X_2,b,id)
- X_3 = aunify(a,X_1,c,X_2) $\vee$ aunify(b,X_2,c,X_2)
- $X_4 = \pi_4$
- X_5 = aunify(a,X_1,d,X_4) $\vee$ aunify(b,X_2,d,X_4)

- Compute lfp using $\pi_4 = \{Xs, Ys\}$

Positive Boolean Formulae

- This domain consists of positive Boolean functions over VI – that returns true when all arguments are true augmented with ***ff***.
- A truth assignment is a model of a Boolean function if the Boolean function is true when arguments are given the true assignment.
- Let assign: Sub→VI→{***tt,ff***}

  assign σ x = tt iff vars(xσ) ={}
- Galois Connection:
  - $\gamma_{Pos}(f) = \{\sigma \mid \text{assign } \sigma \models f \text{ for each } \sigma' \leq \sigma\}$
  - α_{Pos}

Abstract operations

- id = ***tt***
- aunify(a,π_1,b,$_,\pi_2$) =

  let ψ be a tagging substitution ***in***

  let E = mgu(aψ,b) ***in***

  if E=fail then ***ff***

  else $\exists\psi(VI).\ (\alpha_{Pos}(\{E\}) \wedge \pi_1\psi \wedge \pi_2)$
- Let π_4 = ***tt.*** Do the analysis using Pos**.**

OLDT

- OLDT is OLD with tabling. OLD is same as SLD except computational rule always select atom among those that are introduced most recently.
- LD is an OLD.
- T means tabling or memoization: remember all computations and complete duplicate computations by look-up.

Tabling

- Tabling has been known to computer scientists long time ago and has been incorporated into algorithms.
- There are programming language systems that implement tabling – XSB Prolog

Tabling

```
long fib(long n)
{
   if ( n==0 || n==1 )
   then return 1;
   else  return fib(n-1) + fib(n-2);
}
```

Try fib(4)

OLDT

An OLDT structure

- Tree
  - Each node is labeled with a pair of a goal and a substitution
  - Each edge is labeled with a substitution.
  - Look-up nodes (processed by table look-up)
  - Solution nodes (processed as in a SLD tree)
  - Failure nodes
- Table
  - An entry has a key (an atom) and a solution list (instances of the atom)
- Association
  - pointers from look-up nodes to solution lists

Initial OLDT structure

For query (G,σ) and program P;

- Tree – single node labeled with (G,σ)
- Table – empty
- Association – empty

OLDT Resolution

Repeat

- Process a leaf node with non-empty goal
- Process a look-up node

Until no change to the OLDT structure

Process a leaf node $((A,G),\sigma)$

Case -- $A\sigma$ is an variant of a key in the table

- Mark the leaf a look-up node
- Create a child node $(G, \sigma\theta)$ for $((A,G),\sigma)$ (no longer a leaf) for each $A\sigma\theta$ in the solution list for the key and label the edge θ
- make a pointer from $((A,G),\sigma)$ to the tail of the solution list

Process a leaf node $((A,G),\sigma)$

Case -- $A\sigma$ is not an variant of any key in the table

- Mark the leaf a solution node
- Add to the table an entry with key $A\sigma$ and empty solution list
- For each clause (renamed) $H \leftarrow B$ such that $\theta=mgu(H, A\sigma) \neq fail$
  - Create a child node $((B,G), \sigma\theta)$ for $((A,G),\sigma)$ and label the edge θ
  - If B is empty then sub-refutations for several atomic calls have been finished, solution lists of these calls must be updated.
- If there is no such clause, create a child node labeled failure for $((A,G),\sigma)$

Updating solution lists

- Let $A\sigma$ be an atom completed by a unit clause and $\theta_1 \ldots \theta_k$ be substitutions that label edges from the node in which $A\sigma$ is located to the node that is created after the unit clause is applied. Then $A\sigma\theta_1 \ldots \theta_k$ is added to the rear of the solution list for $A\sigma$.

Process a look-up node $((A,G),\sigma)$

If the pointer for the look-up node points to middle of the solution list for a key

- Add a child node $(G, \sigma\theta)$ for $((A,G),\sigma)$ for each $A\sigma\theta$ at or after the pointed position in the solution list for the key and label the edge θ
- position the pointer to the tail of the solution list

Example

arc(a,b).

arc(b,c).

arc(c,a).

path(X,Y) $\leftarrow$ arc(X,Y).

path(X,Z) $\leftarrow$ arc(X,Y),path(Y,Z).

Abstract OLDT

- Substitution → abstract substitution/concrete substitution
- Key → a pair of an atom and an abstract substitution (restricted to variables in the atom)
- Solution → a pair of an atom and an abstract substitution (restricted to variables in the atom)
- Unification → abstract unification
- Composition → abstract composition

Example

append([ ], X, X).

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

$\leftarrow$ append(Xs,Ys,Zs).

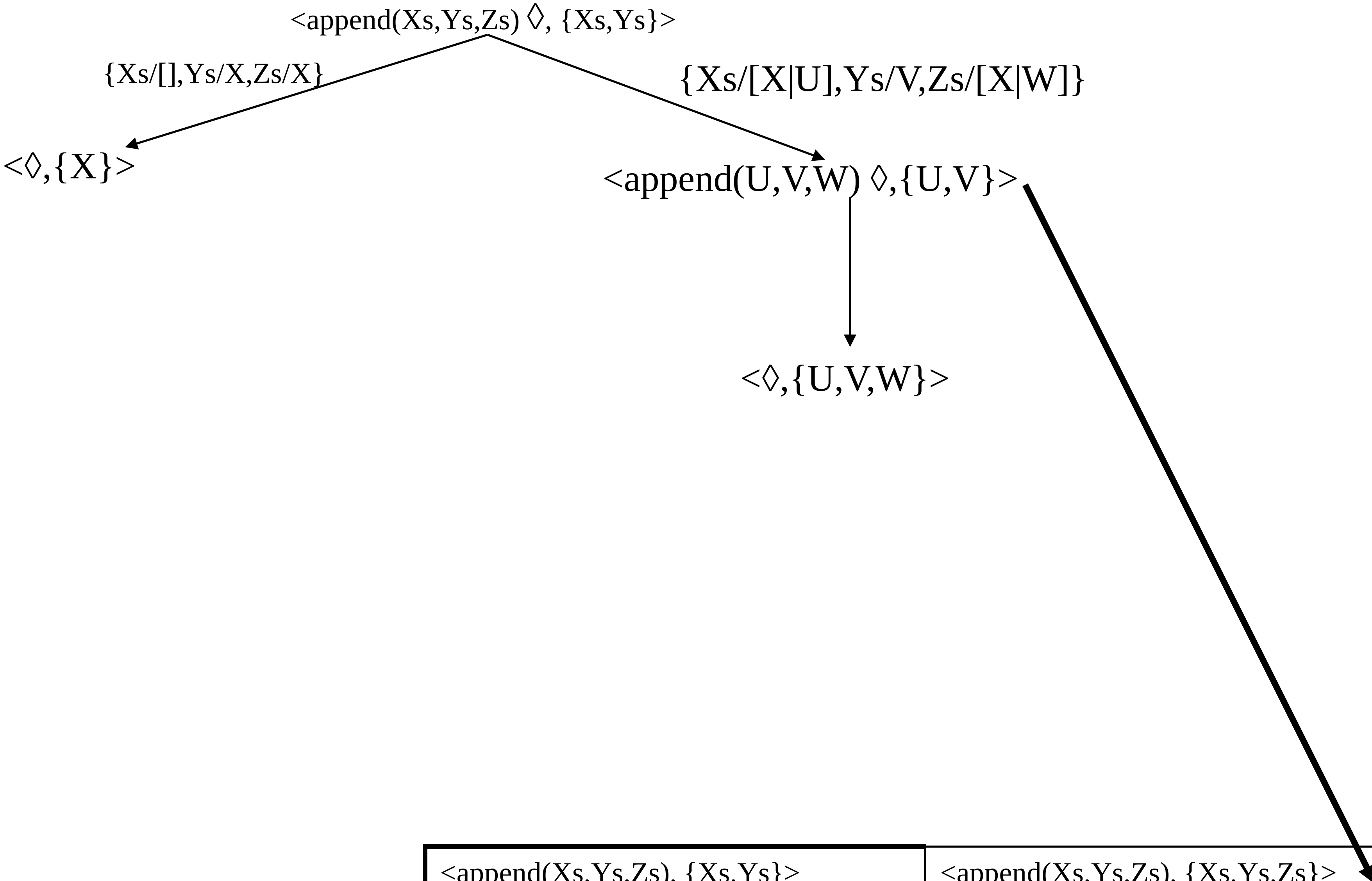
<append(Xs,Ys,Zs), {Xs,Ys}>

<append(Xs,Ys,Zs) ◊, {Xs,Ys}>

{Xs/[],Ys/X,Zs/X}

<◊,{X}>

<append(Xs,Ys,Zs), {Xs,Ys}>	<append(Xs,Ys,Zs), {Xs,Ys,Zs}>



<append(Xs,Ys,Zs), {Xs,Ys}>	<append(Xs,Ys,Zs), {Xs,Ys,Zs}>

Top-down/bottom-up analysis

- Top-down analysis emulates a top-down execution mechanism, like OLDT or SLD.

- Bottom-up analysis emulates a bottom-up execution mechanism such as T_P.

Abstract Compilation

- Apply abstraction to clauses to obtain an abstract program

- Apply T_P to compute success set of the abstract program

Normalized programs

```
append([ ], Ys, Ys).
append([X|Us],Vs,[X|Ws]) ←
    append(Us,Vs,Ws).

append(Xs,Ys,Zs) ← Xs = [ ],Ys=Zs.
append(Xs,Ys,Zs) ←
   Xs=[X|Us], Zs=[X|Ws],
   append(Us,Ys,Ws).
```

Abstraction α_{Pos}

append(Xs,Ys,Zs) ←

  Xs = [ ],

  Ys=Zs.

append(Xs,Ys,Zs) ←

  Xs=[X|Us],
  Zs=[X|Ws],

  append(Us,Ys,Ws).

append(Xs,Ys,Zs) ←

  Xs /\ (Ys↔ Zs).

append(Xs,Ys,Zs) ←

  ∃{X,Us,Ws}.

  Xs ↔ X/\Us /\

  Zs ↔ X/\Ws /\

  append(Us,Ys,Ws).

Abstract atoms

An abstract atom has the following form:

$$p(X_1,X_2,\ldots,X_n) \leftarrow f(X_1,X_2,\ldots X_n)$$

where p is a predicate of arity n.

Success set

append(Xs,Ys,Zs) $\leftarrow$ ***Xs*** /\ ***(Ys***$\leftrightarrow$***Zs)***

append(Xs,Ys,Zs) $\leftarrow$
$\exists$\{X,Us,Ws\}.
Xs $\leftrightarrow$ X/\Us /\
Zs $\leftrightarrow$ X/\Ws /\
Us /\ (Ys $\leftrightarrow$ Ws)

append(Xs,Ys,Zs) $\leftarrow$ ***(Xs/\Ys***$\leftrightarrow$***Zs)***

Quicksort

```
qs(Xs,Ys) ←
    Xs=[],Ys=[].
qs(Xs,Ys) ←
    Xs=[X|Xs1],
    pt(X,Xs1,Ls,Bs),
    qs(Ls,Ls1),
    qs(Bs,Bs1),
    Bs2=[X|Bs1],
    append(Ls1,Bs2,Ys).
```

```
pt(X,Xs,Ls,Bs) ←
    Xs=[],Ls=[],Bs=[].
pt(X,Xs,Ls,Bs) ←
    Xs=[X1|Xs1],
    X < X1,
    pt(X,Xs1,Ls,Bs1),
    Bs = [X1|Bs1].
pt(X,Xs,Ls,Bs) ←
    Xs=[X1|Xs1],
    X >= X1,
    pt(X,Xs1,Ls1,Bs),
    Ls = [X1|Ls1].
```

Abstraction of Quicksort

$qs(Xs,Ys) \leftarrow$

$\quad Xs \wedge Ys$

$qs(Xs,Ys) \leftarrow$

$\quad \exists\{X,Xs1,Ls,Bs,Ls1,Bs1,Bs2\}.$

$\quad Xs \leftrightarrow X \wedge Xs1 \wedge$

$\quad pt(X,Xs1,Ls,Bs) \wedge$

$\quad qs(Ls,Ls1) \wedge$

$\quad qs(Bs,Bs1) \wedge$

$\quad Bs2 \leftrightarrow X \wedge Bs1 \wedge$

$\quad append(Ls1,Bs2,Ys)$

$pt(X,Xs,Ls,Bs) \leftarrow$

$\quad Xs \wedge Ls \wedge Bs$

$pt(X,Xs,Ls,Bs) \leftarrow$

$\quad \exists\{X1,Xs1,Ls,Bs1\}.$

$\quad (Xs \leftrightarrow X1 \wedge Xs1) \wedge$

$\quad X \wedge X1 \wedge$

$\quad pt(X,Xs1,Ls,Bs1) \wedge$

$\quad Bs \leftrightarrow X \wedge Bs1$

$pt(X,Xs,Ls,Bs) \leftarrow$

$\quad \exists\{X1,Xs1,Ls1\}.$

$\quad (Xs \leftrightarrow X1 \wedge Xs1) \wedge$

$\quad X \wedge X1 \wedge$

$\quad pt(X,Xs1,Ls1,Bs),$

$\quad Ls \leftrightarrow X1 \wedge Ls1$

Simplifying Abstract Quicksort

qs(Xs,Ys) $\leftarrow$
    Xs /\Ys
qs(Xs,Ys) $\leftarrow$
    ∃{X,Xs1,Ls,Bs,Ls1,Bs1,Bs2}.
    Xs $\leftrightarrow$X/\Xs1 /\
    pt(X,Xs1,Ls,Bs) /\
    qs(Ls,Ls1) /\
    qs(Bs,Bs1) /\
    Bs2 $\leftrightarrow$ X/\Bs1 /\
    append(Ls1,Bs2,Ys)

pt(X,Xs,Ls,Bs) $\leftarrow$
    Xs/\Ls/\Bs
pt(X,Xs,Ls,Bs) $\leftarrow$
    ∃{Xs1,Ls,Bs1}.
    (Xs $\leftrightarrow$Xs1) /\
    X /\
    pt(X,Xs1,Ls,Bs1) /\
    Bs $\leftrightarrow$ Bs1
pt(X,Xs,Ls,Bs) $\leftarrow$
    ∃{Xs1,Ls,Ls1}.
    (Xs $\leftrightarrow$ Xs1) /\
    X /\
    pt(X,Xs1,Ls1,Bs),
    Ls $\leftrightarrow$Ls1