Static Analysis:
Applications and Logics

David Schmidt

Kansas State University

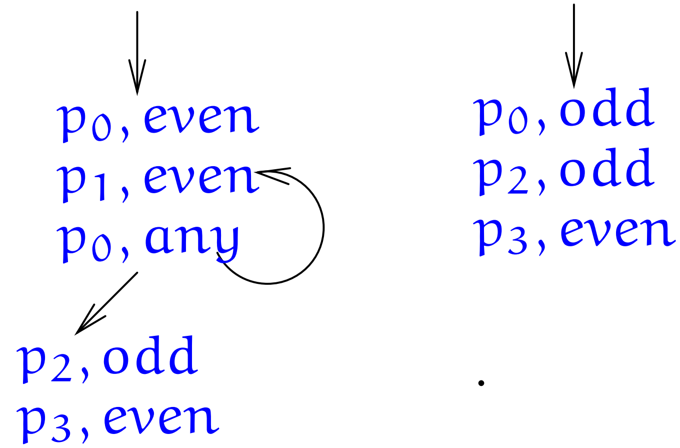
`www.cis.ksu.edu/~schmidt`

Outline

1. Applications:
   - abstract testing and safety checking
   - program transformation
   - assertion checking and discovery
2. Logics:
   - state logics: propositional
   - trace logics: linear- and branching-time temporal logics

Abstract testing and model generation

```
p0 : while isEven(x) {
       p1 : x = x div 2;
     }
p2 : x = 4 * x;
p3 : exit
```



Each trace tree denotes an abstract "test" that covers a set of concrete test cases, e.g., $\gamma(even) = \{..., -2, 0, 2, ...\}$.

Forms of abstract testing:

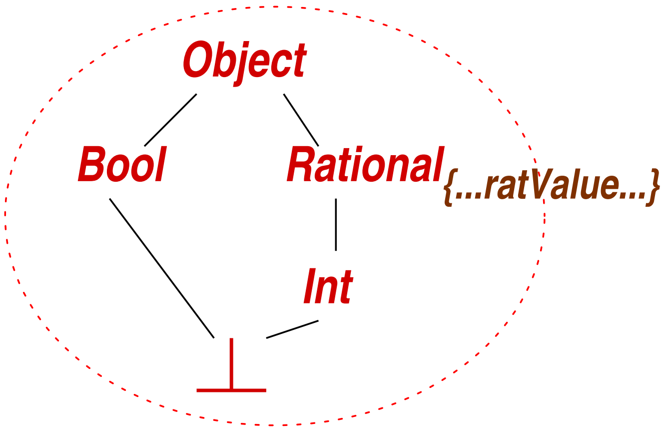
- *Black box*: For each test set, $S \subseteq C$, we abstractly interpret with $\alpha(S) \in A$. *(Best precision: ensure that $S = \gamma(\alpha(S))$.)*
- *White box:* for each conditional, B_i, in the program, ensure there is some $a_i \in A$ such that $\gamma(a_i) = \{s \mid B_i \text{ holds for } s\}$

Once we generate an abstract model, we can analyze it further — ask questions of its paths and nodes — via *model checking*.

Low-level safety checking

There are a variety of errors that might be checked on an abstract model; one example is *type casting*:

p_i : ... ((Rational) x).ratValue()...



Perhaps a static analysis calculates the abstract store arriving at p_i:

- $p_i, \langle ...x : \mathrm{Int}... \rangle$: no error possible — remove the run-time check (because $\mathrm{Int} \sqsubseteq \mathrm{Rational}$, hence $\gamma(\mathrm{Int}) \subseteq_C \gamma(\mathrm{Rational})$).
- $p_i, \langle ...x : \mathrm{Object}... \rangle$: possible error — retain run-time check (because $\mathrm{Object} \not\sqsubseteq \mathrm{Rational}$)
- $p_i, \langle ...x : \mathrm{Bool}... \rangle$: definite error, because $\mathrm{Bool} \sqcap \mathrm{Rational} = \bot$ (assuming $\gamma(\bot) = \bot_C$).

The approach to safety checking:

1. Design a Galois connection, $C\langle\alpha,\gamma\rangle A$, such that all "checkpoint conditions," $c_i \in C$, are abstracted *exactly* by $\alpha(c_i)$ (that is, $c_i = \gamma(\alpha(c_i))$ or equivalently, $c_i \in \gamma[A]$). (Otherwise, we might have that $a' \sqsubseteq_A \alpha(c_i)$ yet $\gamma(a') \not\subseteq_C c_i$.)

2. For each checkpoint, c_i at program point p_i, for each abstract value, a_i, that arrives at p_i, check if $a_i \sqsubseteq_A \alpha(c_i)$.

   If $a_i \sqsubseteq_A \alpha(c_i)$, then no error is possible; if $a_i \not\sqsubseteq_A \alpha(c_i)$, then an error is possible.

When $\gamma(\bot_A) = \bot_C$, and $\bot_C$ denotes no value/dead-code, then $a_i \sqcap \alpha(c_i) = \bot_A$ implies $\gamma(a_i) \cap \gamma(\alpha(c_i)) = \gamma(a_i) \cap c_i = \bot_C$. Thus, no $c \subseteq_C \gamma(a_i)$ satisfies c_i — a definite error.

Two more examples:

Array-bounds and arithmetic over- and under-flow checks

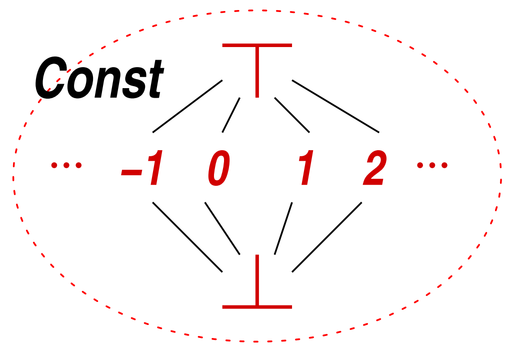
- *Analysis:* interval analysis, where values have form, $[i, j]$, $i \leq j$.
- *Checkpoints:* for `a[e]` — `e` has value in range, $[0,$ `a`$.length]$;
  for `int x = e` — `e` has value in range, $[-2^{31} - 1,\ +2^{31} - 1]$

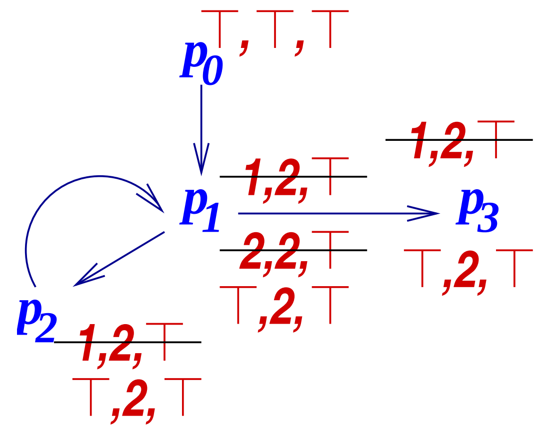
Uninitialized variables, dead-code, and erroneous-state checks

- *Analysis:* constant propagation, where values are $\{k\}$, $\bot$, or $\top$.
- *Checkpoints:*
  uninitialized variables: referenced variables have value $\neq \bot$;
  dead code: at program point p_i, arriving store has value $\neq \bot$;
  erroneous states: at program point $p_i : \mathrm{Error}$, arriving store has value $= \bot$. *(Note:* This can be combined with a *backwards* analysis, starting from each $p_i : \mathrm{Error}$ with store $\top$, working backwards to see if an initial state is reached.*)*

Program transformation: *Constant folding*

```
p0 : x = 1; y = 2;
p1 : while (x < y + z) {
       p2 : x = x + 1;
     }
p3 : exit
```





The analysis tells us to replace y at p_1 by 2:

`x = 1; y = 2; while (x < 2 + z)  x = x + 1`

Basic principle of program transformation:

If $a_i \in A$ *arrives at point* $p_i : S$*, where* $f_i : C \to C$ *is the concrete transfer function, and there are some* S', f' *such that* $f_i(c) = f'(c)$ *for all* $c \sqsubseteq_C \gamma(a_i)$*, then* S *can be replaced by* S' *at* p_i*.*

For constant folding, the transformation criteria are the abstract integers $\ldots - 1, 0, 1, \ldots$ (but not $\top$).

Program transformation: *Code motion*

A compiler translates a program into blocks of three-address code:

```
prod = 0;
i = 1;
do {
   prod = prod + a[i] * b[i];
   i = i + 1;
} while  i <= 20
```

The translation sometimes generates ineffcient code, as array-indexing expressions are expanded:

```
prod = 0
i = 1

L:
t1 = 4 * i
t2 = addr(a) - 4
t3 = t2[t1]
t4 = addr(b) - 4
t5 = t4[t1]
t6 = t3 * t5
prod = prod + t6
i = i + 1
if i <= 20 goto L
```

Note: This example comes from the Aho and Ullman "green dragon" compiling text.

A reaching-definitions analysis helps calculate that the statements in the loop's body that assign to `t2` and `t4` are constant — the assignments can be moved out of the loop:

```
prod = 0
i = 1

L:
t1 = 4 * i
t2 = addr(a) - 4
t3 = t2[t1]
t4 = addr(b) - 4
t5 = t4[t1]
t6 = t3 * t5
prod = prod + t6
i = i + 1
if i <= 20 goto L
```

```
prod = 0
i = 1
t2 = addr(a) - 4
t4 = addr(b) - 4

L:
t1 = 4 * i
t3 = t2[t1]
t5 = t4[t1]
t6 = t3 * t5
prod = prod + t6
i = i + 1
if i <= 20 goto L
```

Precondition checking and assertion synthesis

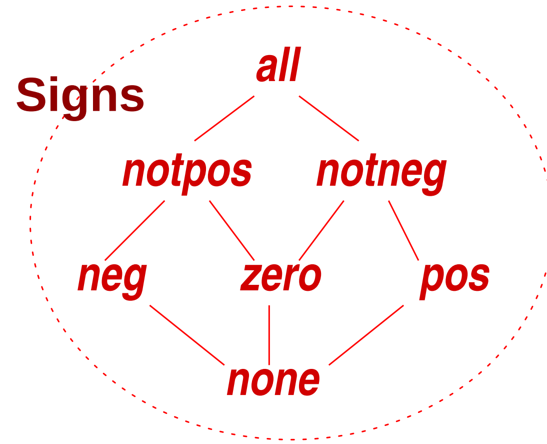
A backwards-necessarily analysis can synthesize precondition assertions that ensure achievement of a postcondition:

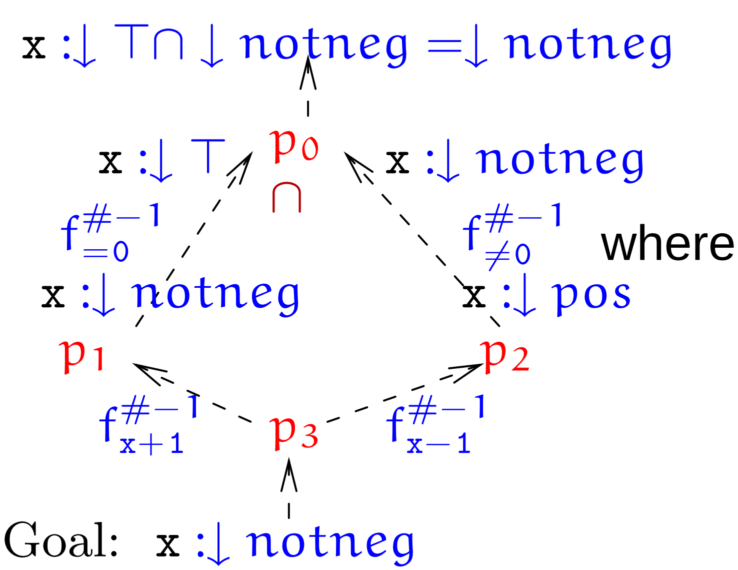
p_0: `if x=0`

$\quad p_1$: `then x = x+1`

$\quad p_2$: `else x = x-1`

p_3 : halt $\quad \langle \mathtt{x} :\downarrow \mathrm{notneg} \rangle$





$f^{\#}_{=0}(a) = a \sqcap \mathrm{zero} = \alpha \circ f_{=0} \circ \gamma$

$f^{\#}_{\neq 0}(a) = \alpha \circ f_{\neq 0} \circ \gamma$, e.g., $f^{\#}_{\neq 0}(\mathrm{notneg}) = \mathrm{pos}$;

$\qquad f^{\#}_{\neq 0}(\mathrm{zero}) = \bot; f^{\#}_{\neq 0}(\top) = \top$

$f^{\#}_{+1}(a) = \alpha \circ f_{+1} \circ \gamma$, e.g., $f^{\#}_{+1}(\mathrm{notneg}) = \mathrm{pos}$

$\downarrow a = \{a' \in A \mid a' \sqsubseteq a\}$

$f^{\#-1}(S) = \{a \in A \mid f^{\#}(a) \in S\}$

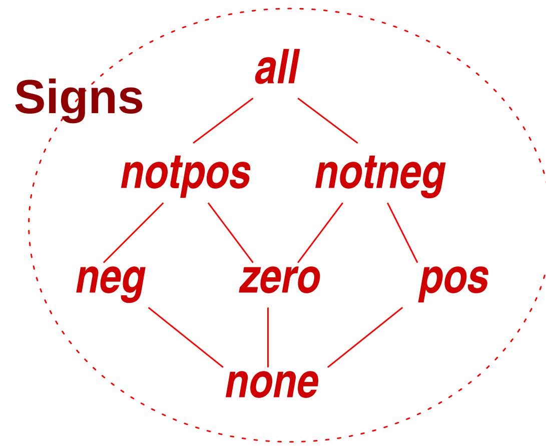
The entry condition can be used with a *forwards-possibly* analysis to generate postconditions that sharpen the assertions:

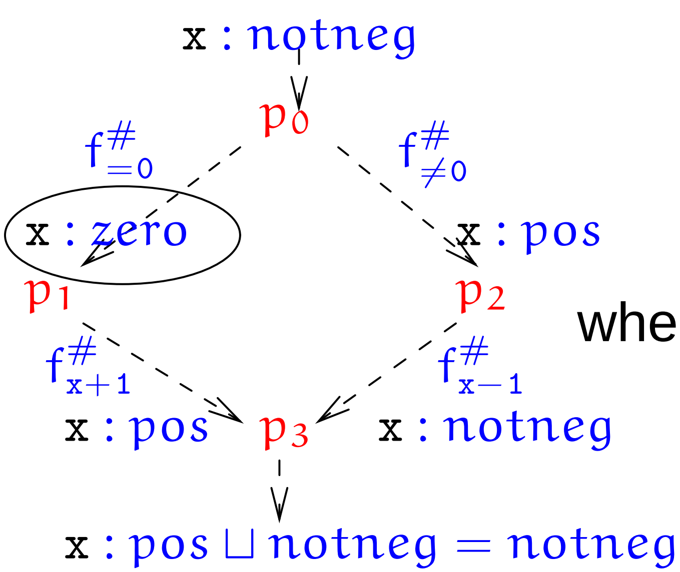
$\langle x : notneg \rangle$ p_0: `if x=0`

p_1 : `x = x+1`

p_2 : `x = x-1`

p_3 : halt





where

$f^{\#}_{=0}(a) = a \sqcap zero = \alpha \circ f_{=0} \circ \gamma$

$f^{\#}_{\neq 0}(a) = \alpha \circ f_{\neq 0} \circ \gamma$, e.g., $f^{\#}_{\neq 0}(notneg) = pos$;

$f^{\#}_{\neq 0}(zero) = \bot$; $f^{\#}_{\neq 0}(\top) = \top$

$f^{\#}_{+1}(a) = \alpha \circ f_{+1} \circ \gamma$, e.g., $f^{\#}_{+1}(notneg) = pos$

The forwards-backwards analyses can be repeatedly alternated.

General assertion checking

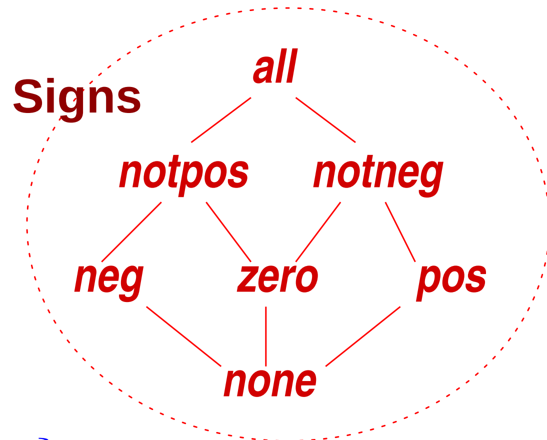
Checking user-supplied assertions is a form of low-level safety checking, e.g., we might check $p_i : \texttt{assert \{ x != 0 \}}$

It is crucial that an assertion, $\phi \in C$ (say, for $C = \wp(Store)$), be *exactly abstracted*: $\phi \in \gamma[A]$, that is, $\gamma(\alpha(\phi)) = \phi$.

Example: `x != 0`, that is,

$\texttt{x} \in \{... - 2, -1, 1, 2, ...\}$

is not exactly abstracted in **Signs**:



Let ϕ abbreviate $\{s : Store \mid \phi$ holds for $s\}$:

$\alpha(\texttt{x != 0}) = \alpha(\texttt{x} < 0 \vee \texttt{x} > 0) = \alpha(\texttt{x} < 0) \sqcup \alpha(\texttt{x} > 0) = neg \sqcup pos = all$.

To check `x!=0` for a, we should check if $a \sqsubseteq neg$ **or** $a \sqsubseteq pos$ — *not* $a \sqsubseteq all$! (The underlying issue is $\gamma(neg \sqcup pos) \neq \gamma(neg) \cup \gamma(pos)$.)

Nonetheless, the **Signs** domain defines its own "logic."

An abstract domain defines a "logic"

For $\wp(D)\langle\alpha,\gamma\rangle A$, each $a \in A$ is a "property/predicate," and $\gamma(a) \in \wp(D)$ defines those concrete states that make a "true":

$$s \ \mathit{has}\ a \text{ , written } s \models_A a, \textbf{ iff } s \in \gamma(a)$$

Because γ preserves $\sqcap$, we have that

$$c \models a \sqcap b \textbf{ iff } c \models a \text{ and } c \models b$$
$$a' \sqsubseteq a \sqcap b \textbf{ implies } \text{for all } c \in D, \text{ if } c \models a' \text{ then } c \models a \textbf{ and } c \models b$$

$\sqcap$ behaves like conjunction — $\langle A, \sqcap\rangle$ is a logic.

Example: From $0 \models_{Sign} \mathsf{notneg}$ and $0 \models_{Sign} \mathsf{notpos}$, we deduce $0 \models_{Sign} \mathsf{notneg} \sqcap \mathsf{notpos}$.

When $\phi_1 \in \gamma[A]$ and $\phi_2 \in \gamma[A]$, we safety-check whether $a \in A$ satisfies $\mathtt{assert}\{\phi_1 \wedge \phi_2\}$ by checking $a \sqsubseteq_A \alpha(\phi_1)$ *and* $a \sqsubseteq_A \alpha(\phi_2)$.

Example: We check $\langle \mathtt{x} : \mathit{zero}\rangle \mathtt{assert}\ \{\mathtt{x} \leq 0 \wedge \mathtt{x} \geq 0\}$ by checking $\mathit{zero} \sqsubseteq_{Sign} \alpha(\mathtt{x} \leq 0) = \mathsf{notpos}$ *and* $\mathit{zero} \sqsubseteq_{Sign} \alpha(\mathtt{x} \geq 0) = \mathsf{notneg}$.

Constructing the abstract logic, $\langle A, \sqcap \rangle$

We can construct the logic, $\langle A, \sqcap \rangle$, and its Galois connection as a freely generated complete lattice:

Say the concrete domain is D, and F is a *possibly infinite* set of primitive "properties." Say that we have an entailment relation, $\models \subseteq D \times F$.

Example: $D = Int$ and $F = Sign = \{neg, notneg, notpos, pos\}$; $-2 \models notpos$, $0 \models notneg$, etc.

We generate a Moore family from F by constructing all possible conjunctions of the properties listed in F:

Define $\underline{F}$ to be the set of conjunctions of form, $\sqcap_{i \in I} a_i$, where $I \subseteq Nat$ and all $a_i \in F$

Example: $notpos \sqcap notneg \in \underline{Sign}$.

Next, we interpret the conjunctions with the map, $\gamma : \underline{F} \rightarrow \wp(D)$:

$$\gamma(\sqcap_{i\in I} a_i) = \bigcap_{i\in I} \delta(a_i)$$

$$\texttt{where } \delta(a) = \{c \in D \mid c \models a\}$$

$\gamma(a_1 \sqcap ... \sqcap a_n)$ gives all $d \in D$ such that "$d \models a_1 \sqcap ... \sqcap a_n$".

Example: $\gamma(notpos \sqcap notneg) = \{..., -1, 0\} \cap \{0, 1, ...\} = \{0\}$.

$\gamma[\underline{F}]$ is a *Moore family*, because it is closed under intersections: $\bigcap_{i\in I} \gamma(f_i) = \gamma(\sqcap_{i\in I} f_i)$. *There is a Galois connection,* $\wp(D)\langle\alpha, id\rangle\gamma[\underline{F}]$.

Since $\sqcap$ is associative, commutative, and absorptive, we might try representing $\sqcap_{i \in I} a_i$ as $\{a_i\}_{i \in I}$, e.g., $\mathit{notpos} \sqcap \mathit{notneg}$ is represented as $\{\mathit{notpos}, \mathit{notneg}\}$. This means $\sqcap$ is just set union.

We have this Galois connection, $(\wp(D), \subseteq)\langle \alpha, \gamma \rangle(\wp(F), \supseteq)$*:*

$$\gamma(T) = \bigcap_{a \in T} \delta(a)$$
$$\alpha(S) = \{a \mid S \subseteq \delta(a)\}$$
$$\textit{where } \delta(a) = \{c \in D \mid c \models a\}$$

Here, $S \sqsubseteq T$ iff $T \subseteq S$, which is not so interesting.

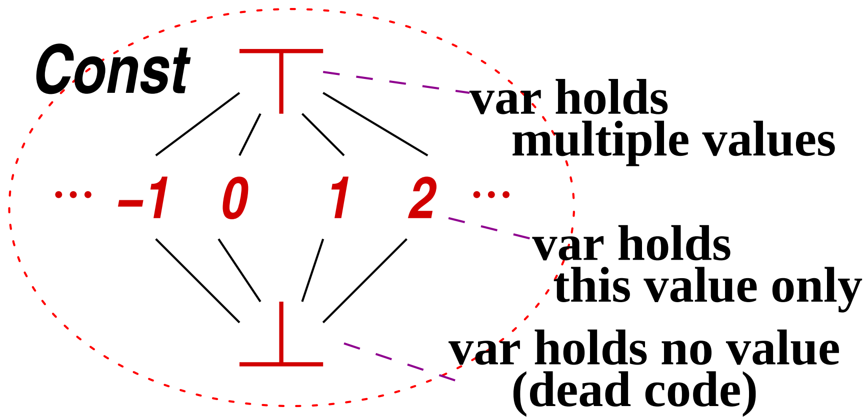
If the properties, F, are *partially ordered* and $\delta : F \to \wp(D)$ is *monotone*, then we have this Galois connection:

$$(\wp(D), \subseteq)\langle \alpha, \gamma \rangle(\{\uparrow T \mid T \subseteq F\}, \supseteq),$$

where $\uparrow T = \{a' \in F \mid \text{exists } a \in T, a \sqsubseteq_F a'\}$ is the *up-closure* of set T. This gives us more interesting "implications," $\sqsubseteq$, in the abstract domain.

Making $\sqcup$ into disjunction (1)

The $\sqcup$ operation does *not* behave like "or" for the ***Const*** abstract domain:



For example, we have $1 \models 1$, and also $1 \models \top$. But

$$\top = 1 \sqcup 2 = 1 \sqcup 2 \sqcup 3 = 2 \sqcup 3, \text{ etc.}$$

This implies $1 \models 2 \sqcup 3$. But is it true that $1 \models 2$ *or* $1 \models 3$? **No.**

The technical problem is that $\gamma(a \sqcup b) \neq \gamma(a) \cup \gamma(b)$. The problem should be repaired by inserting a more precise element than $\top$ to denote $2 \sqcup 3$, etc.

Making $\sqcup$ into disjunction (2)

Given $\wp(D)\langle\alpha,\gamma\rangle A$, we have that $\sqcup$ behaves like disjunction when

$$\gamma(a \sqcup_A b) = \gamma(a) \cup \gamma(b)$$

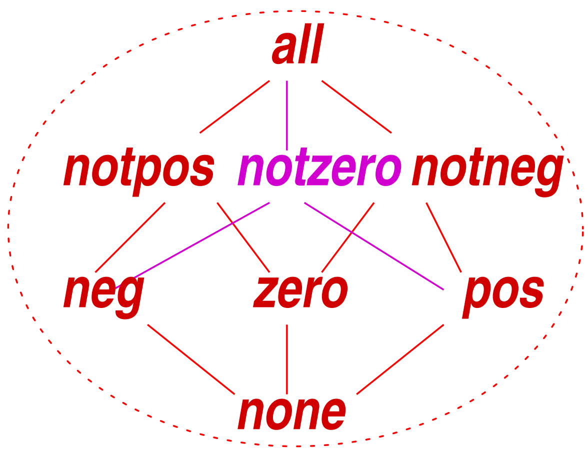
That is, $\sqcup_A$ must be *forwards complete* for $\cup$. We then have

- $c \models a \sqcup b$ **iff** $c \models a$ or $c \models b$
- $a' \sqsubseteq a \sqcup b$ **implies** for all $c \in D$, if $c \models a'$ then $c \models a$ or $c \models b$.

and $\langle A, \sqcap, \sqcup\rangle$ **is a logic.**

When there are $a, b \in A$ such that $\gamma(a \sqcup_A b) \neq \gamma(a) \cup \gamma(b)$, we insert a new element, a', such that $a' = a \sqcup b$ and $\gamma(a \sqcup_A b) = \gamma(a) \cup \gamma(b)$.

Example: The completed **Signs** domain: It is precise enough to check the assertion, `x!=0`, that is, `x<0 or x>0`



Making $\sqcup$ into disjunction (3): Disjunctive completion

Given abstract domain, $(A, \sqsubseteq_A)$, we can construct its *disjunctive completion* as

$$\wp_\downarrow(A) = (\{\downarrow S \mid S \subseteq A\}, \subseteq)$$

where $\downarrow S = \{a \in A \mid \text{exists } a' \in S, a \sqsubseteq_A a'\}$. That is, $\downarrow S$ is the *down closure* of S.

Intuition: $\downarrow\{a\} \in \wp_\downarrow(A)$ represents $a \in A$. A "non-singleton" set, $\downarrow\{a_0, ..., a_i, ...\}$, represents the join of elements $\{a_0, ..., a_i, ...\} \subseteq A$.

Given the Galois connection, $\wp(D)\langle\alpha, \gamma\rangle A$, we can construct the Galois connection on $\wp_L(A)$ as $\wp(D)\langle\alpha_\downarrow, \gamma_\downarrow\rangle\wp_\downarrow(A)$:

$$\gamma_\downarrow(T) = \bigcup_{a \in T} \gamma(a)$$
$$\alpha_\downarrow(S) = \bigcap\{T \in \wp_L(A) \mid S \subseteq \gamma_\downarrow(T)\}$$

and we can prove easily that this Galois connection makes $\sqcup_{\wp_\downarrow(A)} = \cup$ (in $\wp_\downarrow(A)$) forwards complete for $\cup$ (in $\wp(D)$).

Constructing an abstract logic, $\langle A, \sqcap, \sqcup \rangle$

For concrete domain D, a *finite* set of "facts," F, and an entailment relation, $\models \subseteq D \times F$, let $\overline{F}$ be the set of finite *disjunctive normal form* (*DNF*) phrases built from F:

$$(a_{11} \sqcap a_{12} \sqcap ...a_{1n}) \sqcup (a_{21} \sqcap a_{22} \sqcap ...a_{2n}) \sqcup \cdots \sqcup (a_{m1} \sqcap a_{m2} \sqcap ...a_{mn})$$

for all $a_{ij} \in F$ and $m, n \geq 0$.

Define this map:

$$\gamma(a) = \{c \in D \mid c \models a\}$$

$$\gamma(a \sqcup b) = \gamma(a) \cup \gamma(b)$$

$$\gamma(a \sqcap b) = \gamma(a) \cap \gamma(b)$$

This is of course the distributive complete lattice that is freely generated from generator set, D.

1. $\gamma[\overline{F}]$ is closed under unions: $\gamma(f_1) \cup \gamma(f_2) = \gamma(f_1 \sqcup f_2)$, and $f_1 \sqcup f_2$ is in DNF.

2. As F is finite, $\gamma[\overline{F}]$ is a *Moore family*: $\gamma(f_1) \cap \gamma(f_2) = \gamma(f_1 \sqcap f_2)$, and since $f \sqcap (g \sqcup h) \equiv_\gamma (f \sqcap g) \sqcup (f \sqcap h)$, there is a DNF formula that is γ-equivalent to $f_1 \sqcap f_2$.

The logic for the Galois connection $\wp(D)\langle\alpha, id\rangle\gamma(\overline{F})$ is $\langle \overline{F}, \sqcap, \sqcup \rangle$.

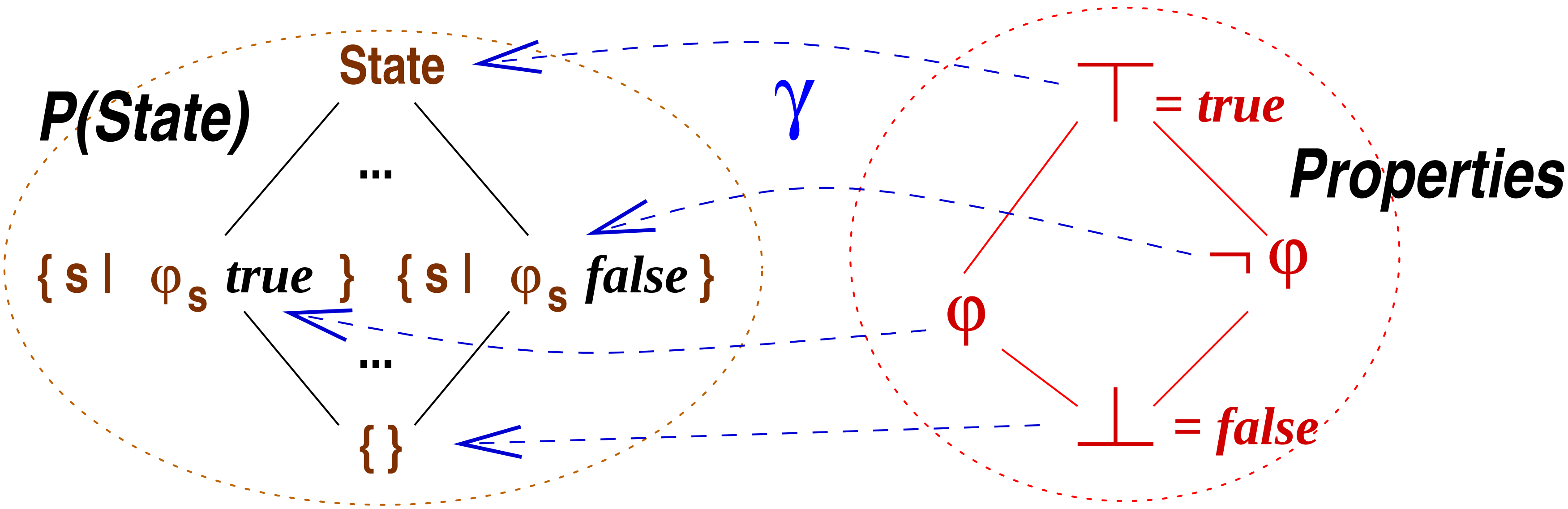
Example: Generating the Sign logic, $\langle \overline{Sign}, \sqcap, \sqcup \rangle$:

From $Sign = \{neg, zero, pos\}$ and $\models \subseteq Int \times Sign$, we obtain

Sign = $\overline{Sign}/\gamma$ =

$$
\begin{tikzcd}
 & (\sqcap()) \equiv neg \sqcup zero \sqcup pos \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
neg \sqcup zero \arrow[d, dash] \arrow[dr, dash] & neg \sqcup pos \arrow[dl, dash] \arrow[dr, dash] & zero \sqcup pos \arrow[dl, dash] \arrow[d, dash] \\
neg \arrow[dr, dash] & zero \arrow[d, dash] & pos \arrow[dl, dash] \\
 & neg \sqcap zero \sqcap pos &
\end{tikzcd}
$$

Domains might employ a *negation* operation



Each element, a has a *unique* complement element, $\neg a$, such that $a \sqcup \neg a = \top$ and $a \sqcap \neg a = \bot$.

When the domain is a distributive lattice, we have a *boolean algebra*, where these laws hold: $\neg(a \sqcup b) = \neg a \sqcap \neg b$ and $\neg(a \sqcap b) = \neg a \sqcup \neg b$.

If $\wp(D)$ and A are boolean algebras, and γ preserves negation, that is, $\gamma(\neg a) = \sim\gamma(a)$, **then** γ also preserves $\sqcup$.

This makes $(A, \neg, \sqcup, \sqcap)$ *a classical propositional logic.*

Predicate abstraction, revisited

Recall that we proved $z \geq x \wedge z \geq y$ at p_3:

```
p0 :  if x < y
p1 :    then z = y
p2 :    else z = x
p3 :  exit
```

$$\begin{tikzcd}[column sep=small] & p_0, \langle ?, ?, ? \rangle \arrow[dl] \arrow[dr] & \\ p_1, \langle t, ?, ? \rangle \arrow[d] & & p_2, \langle f, ?, ? \rangle \arrow[d] \\ p_3, \langle t, t, t \rangle & & p_3, \langle f, t, t \rangle \end{tikzcd}$$

We chose three predicates, and computed their values at the program's points. At p_3, $\phi_2 \wedge \phi_3$ holds.

$$\phi_1 = x < y$$

$$\phi_2 = z \geq x$$

$$\phi_2 = z \geq y$$

The analysis used a logic, $\langle \tilde{F}, \sqcap \rangle$, where $\tilde{F}$ is the set of conjunctions generated from $F = \{x < y, z \geq x, z \geq y, \neg x < y, \neg z \geq x, \neg z \geq y\}$.

E.g., $\langle f, ?, ? \rangle$ at p_2 is $\neg\phi_1$, and $\langle f, t, t \rangle$ at p_3 (right) is $\neg\phi_1 \sqcap \phi_2 \sqcap \phi_3$.

Depending on the underlying "logic" and static analysis, there are many forms of "predicate abstraction" and "refinement."

Properties of execution traces: *L*inear *T*ime *L*ogic

We now move from describing properties of states to describing entire execution traces:

p_0 : `while x > 0 {`
  p_1 : *use resource*
    `x = x + 1;  }`
p_2 : *sleep forever*

||

q_0 : `x = 0;`
q_1 : *use resource forever*

$$\pi = \begin{array}{l} p_0, q_0, 1 \\ p_1, q_0, 1 \\ p_0, q_0, 2 \\ p_1, q_0, 2 \\ \qquad p_1, q_1, 0 \\ p_0, q_1, 1 \\ p_1, q_1, 1 \\ p_0, q_1, 2 \\ \vdots \\ p_0, q_1, k \\ \vdots \end{array}$$

Will π enter p_1 in its Future? $\pi \models \mathrm{F}\ \mathrm{at}\ p_1$ — **Yes**

Is it Generally true that p_1 is reached in the Future ? $\pi \models \mathrm{GF}\ \mathrm{at}\ p_1$ — **Yes**

Will π Generally avoid resource misuse? $\pi \models \mathrm{G}\ \neg(\mathrm{at}\ p_1 \sqcap \mathrm{at}\ q_1)$ — **No**

An LTL property, ϕ, describes a pattern of states in a trace:

Let Σ be the concrete states, assume $\wp(\Sigma)\langle\alpha,\gamma\rangle A$, and let $a \in A$:

MiniLTL: $\phi ::= a \mid G\phi \mid F\phi$ Semantics: $[\![\phi]\!] \subseteq \wp(\mathrm{Trace}(\Sigma))$

$[\![a]\!] = \{\pi \mid \pi_0 \in \gamma(a)\}$ (initial state, π_0, has state-property a)

$[\![G\phi]\!] = \{\pi \mid \forall i \geq 0, \pi \downarrow i \in [\![\phi]\!]\}$ (all subtraces of π have ϕ)

$[\![F\phi]\!] = \{\pi \mid \exists i \geq 0, \pi \downarrow i \in [\![\phi]\!]\}$ (exists a subtrace of π with ϕ)

where, for $\pi = s_0 \to s_1 \to \cdots$, let $\pi_0 = s_0$ and $\pi \downarrow i = s_i \to s_{i+1} \to \cdots$.

For $\pi \in \mathrm{Trace}(\Sigma)$, we write $\pi \models \phi$ to assert that $\pi \in [\![\phi]\!]$.

MiniLTL abstracts trace sets: Using $\sqcap$-completion, we build the Galois connection, $(\mathcal{P}(\mathrm{Trace}(\Sigma)), \subseteq)\langle\alpha,\gamma\rangle(\mathcal{P}(\mathrm{MiniLTL}), \supseteq)$, where $\sqcap = \cup$ in $\mathcal{P}(\mathrm{MiniLTL})$:

$\gamma(P) = \bigcap\{[\![\phi]\!] \mid \phi \in P\}$ (the traces that have all the properties in P)

$\alpha(S) = \{\phi \mid S \subseteq [\![\phi]\!]\}$ (properties held by all traces in S)

What we have just accomplished:

- We defined an entailment relation, $\pi \models \phi$ (asserts that $\pi \in [\![\phi]\!]$)
- We constructed a Moore family by closing the entailment relation under conjunction, where $\{a_0, a_1, ..., a_n\}$ denotes $a_1 \sqcap a_2 \sqcap ... \sqcap a_n$.
- The Galois connection, $(\mathcal{P}(\mathrm{Trace}), \subseteq)\langle\alpha, \gamma\rangle(\mathcal{P}(\text{MiniLTL}), \supseteq)$, uses the logic,

  $$\langle\wp(\text{MiniLTL}), \sqcap\rangle, \quad \text{where } \sqcap \text{ is set union.}$$

  For example, $Fa \sqcap Fb$ is $\{Fa\} \cup \{Fb\} = \{Fa, Fb\}$.

The MiniLTL logic is a *linear-time logic* because it expresses properties of “linear” traces.

MiniLTL is a weak logic — it cannot express disjunction (e.g., $Fa \sqcup Fb = \{Fa\} \cap \{Fb\} = \{\} = \top$), nor can it express “until” properties. Nonetheless, we develop it further to learn some standard abstractions on temporal logics.

Abstracting traces

Let Σ be the set of concrete states; the set of concrete traces (sequences of states) is $\mathrm{Trace}(\Sigma) = \Pi_{i \geq 0}\Sigma$.

Say that Σ is abstracted to abstract states: $\wp(\Sigma)\langle\alpha, \gamma\rangle A$. Then, the set of abstract traces is $\mathrm{Trace}(A) = \Pi_{i \geq 0}A$. Let $\kappa \in \mathrm{Trace}(A)$.

We can define $\delta_{\mathrm{Trace}} : \mathrm{Trace}(A) \to \wp(\mathrm{Trace}(\Sigma))$ as

$$\delta_{\mathrm{Trace}}(\kappa) = \{\langle s_i \rangle_{i \geq 0} \mid s_i \in \gamma(\kappa_i)\}$$

That is, $\delta_{\mathrm{Trace}}(\kappa)$ concretizes abstract trace κ to all traces, π, such that $\pi_i \in \gamma(\kappa_i)$, for all $i \geq 0$.

There is also this Galois connection between sets of concrete traces and individual abstract traces:

$$(\wp(\mathrm{Trace}(\Sigma)), \subseteq)\langle\alpha_{\mathrm{Trace}}, \delta_{\mathrm{Trace}}\rangle(\mathrm{Trace}(A), \sqsubseteq_\omega)$$
$$\alpha_{\mathrm{Trace}}(S) = \langle \alpha\{\pi_i \mid \pi \in S\}\rangle_{i \geq 0}$$

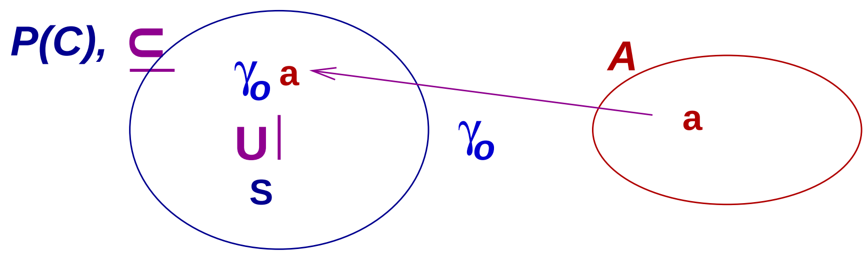
Example: Given the **Parity** abstraction ($even$, odd, $none$, any), and a program with program points p_0, p_1, etc. we might have this abstract trace:

$$\kappa_0 = p_0, even \rightarrow p_1, even \rightarrow p_2, odd \rightarrow ...$$

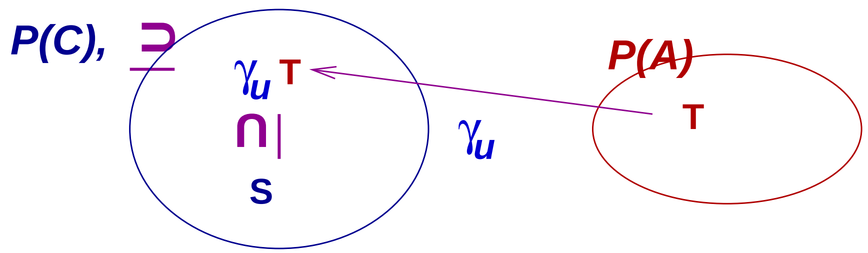
Then, $\delta_{\mathrm{Trace}}(\kappa_0) \subseteq \mathrm{Trace}(\Sigma)$ is a set that includes concrete traces like $p_0, 0 \rightarrow p_1, 2 \rightarrow p_2, 1 \rightarrow ...$ and $p_0, 6 \rightarrow p_1, 6 \rightarrow p_2, 3 \rightarrow ...$ and infinitely many others.

Under-approximation and assertion checking

Over-approximation calculates a *superset* of the concrete answers:



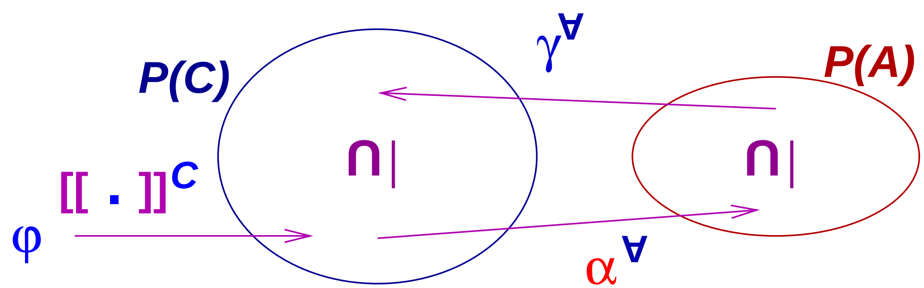
A form of *under-approximation* calculates a *subset* of the answers:



We ***over-approximate*** the answer set, $S \subseteq C$, by some $a \in A$, so that $S \subseteq \gamma_o(a)$, and we ***under-approximate*** an assertion, $[\![\phi]\!]^C \subseteq C$, by some set, $[\![\phi]\!]^A \subseteq A$, so that $\gamma_u[\![\phi]\!]^A \subseteq [\![\phi]\!]^C$. This gives us

$$a \in [\![\phi]\!]^A \text{ implies } c \in [\![\phi]\!]^C\text{, for all } c \in \gamma(a)$$

We define $[\![\phi]\!]^A$ as this under-approximation:



That is,

$[\![\cdot]\!]^A = \alpha^{\forall} \circ [\![\cdot]\!]^C$

so that, for all ϕ:

$[\![\phi]\!]^C \supseteq \gamma^{\forall}[\![\phi]\!]^A$

Given $\delta : A \to \wp(C)$, define $(\wp(C), \supseteq)\langle\alpha^{\forall}_{\delta}, \gamma^{\forall}_{\delta}\rangle(\wp(A), \supseteq)$ as

$$\gamma^{\forall}_{\delta}(T) = \bigcup_{a \in T} \delta(a)$$
$$\alpha^{\forall}_{\delta}(S) = \{a \mid \delta(a) \subseteq S\}$$

This is called the *universal abstraction*. δ is usually the upper adjoint, γ, of the Galois connection, $\wp(C)\langle\alpha, \gamma\rangle A$, but it is not required.

This construction *generalizes* the earlier slide on "General Assertion Checking" — now, we need not require that $[\![\phi]\!]^C$ is a fixed point — $\gamma(\alpha[\![\phi]\!]^C) = [\![\phi]\!]^C$ — to check that $a \sqsubseteq_A \alpha[\![\phi]\!]^C$. *Instead, we check that $a \in [\![\phi]\!]^A$, which is always sound.*

Deriving MiniLTL for abstract traces

Here again is the concrete MiniLTL semantics, $[\![\phi]\!]^{\Sigma} \subseteq \mathrm{Trace}(\Sigma)$, better structured:

$$[\![a]\!]^{\Sigma} = \{\pi \in \mathrm{Trace}(\Sigma) \mid \pi_0 \in \gamma(a)\}$$

$$[\![G\phi]\!]^{\Sigma} \;=\; \mathrm{gen}^{\Sigma}[\![\phi]\!]^{\Sigma}$$

$$\mathrm{gen}^{\Sigma}(M) = \{\pi \in \mathrm{Trace}(\Sigma) \mid \forall i \geq 0, \pi \downarrow i \in M\}$$

$$[\![F\phi]\!]^{\Sigma} = \mathrm{fut}^{\Sigma}[\![\phi]\!]^{\Sigma}$$

$$\mathrm{fut}^{\Sigma}(M) = \{\pi \in \mathrm{Trace}(\Sigma) \mid \exists i \geq 0, \pi \downarrow i \in M\}$$

Here is the expected MiniLTL semantics for checking properties of abstract traces: $[\![\phi]\!]^{A} \subseteq \mathrm{Trace}(A)$

$$[\![\phi]\!]^{A} = \alpha^{\forall}[\![\phi]\!]^{\Sigma}$$

But we prefer to define $[\![\phi]\!]^{A}$ without explicit reference to $[\![\phi]\!]^{\Sigma}$, and a proof by induction on the structure of ϕ shows that $\alpha^{\forall}[\![\phi]\!]^{\Sigma}$ equals the following:

$$[\![a]\!]^A = \alpha^\forall\{\pi \in \mathrm{Trace}(\Sigma) \mid \pi_0 \in \gamma(a)\}$$

$$[\![G\phi]\!]^A \;=\; \alpha^\forall \circ \mathrm{gen}^\Sigma \circ \gamma^\forall([\![\phi]\!]^A)$$

$$[\![F\phi]\!]^A = \alpha^\forall \circ \mathrm{fut}^\Sigma \circ \gamma^\forall([\![\phi]\!]^A)$$

(When using the inductive format, we always have $\alpha^\forall[\![\phi]\!]^\Sigma \sqsubseteq [\![\phi]\!]^A$. When all $[\![\phi]\!]^\Sigma = \gamma^\forall(\alpha^\forall[\![\phi]\!]^\Sigma)$ —fi xed points! —we have equality, as is the case here.)

Further analysis of the embedded functions gives us

$$[\![a]\!]^A = \{\kappa \in \mathrm{Trace}(A) \mid \kappa_0 \sqsubseteq a\}$$

$$[\![G\phi]\!]^A \;=\; \mathrm{gen}^A[\![\phi]\!]^A$$

$$\mathrm{gen}^A(M) = \{\kappa \in \mathrm{Trace}(A) \mid \forall i \geq 0, \kappa \downarrow i \in M\}$$

$$[\![F\phi]\!]^A = \mathrm{fut}^A[\![\phi]\!]^A$$

$$\mathrm{fut}^A(M) = \{\kappa \in \mathrm{Trace}(A) \mid \exists i \geq 0, \kappa \downarrow i \in M\}$$

Because $[\![\phi]\!]^\Sigma \supseteq \gamma^\forall(\alpha^\forall[\![\phi]\!]^\Sigma) = \gamma^\forall[\![\phi]\!]^A$, for all ϕ, we have soundness of trace checking:

Theorem: For $\kappa \in Trace(A)$, $\kappa \in [\![\phi]\!]^A$ implies $\pi \in [\![\phi]\!]^\Sigma$, for all $\pi \in \gamma^\forall\{\kappa\} = \delta(\kappa)$.

That is, if an abstract trace, κ, has ϕ, then so do all the concrete traces it models. (The theorem also holds for a *set*, T of traces such that $T \subseteq [\![\phi]\!]^A$.)

For state $s \in \Sigma$, we write

$$s \models^\Sigma \forall\phi \quad \text{to assert} \quad \{\pi \in Trace(\Sigma) \mid \pi_0 = s\} \subseteq [\![\phi]\!]^\Sigma$$

(similarly for $a \in A$ and $a \models^A \forall\phi$). That is, all traces starting with s have property ϕ.

By the above theorem, we have this result, which justifies linear-time model checking on programs and their start states:

Corollary: $a \models^A \forall\phi$ implies $s \models^\Sigma \forall\phi$, for all $s \in \gamma(a)$.

Generating traces from small-step semantics

A trace is generated from a program, P. Say that τ^Σ is the small-step semantics that generates traces for P, and say that $\mathrm{Trace}(\tau^\Sigma)$ is the set of all possible traces generated from τ^Σ using all states in Σ as starting states.

Let $\mathrm{Trace}(\tau^\Sigma) \Downarrow s$ denote the subset of $\mathrm{Trace}(\tau^\Sigma)$ holding exactly all traces starting with s.

In general, for a set $T \subseteq \mathrm{Trace}(\tau^\Sigma)$, defi ne$T \Downarrow s = \{\pi \in T \mid \pi_0 = s\}$

We can use a *state* to abstract a set of *traces* — we use $s \in \Sigma$ to abstract the set, $\mathrm{Trace}(\tau^\Sigma) \Downarrow s$. This simple idea lies at the heart of *branching-time model checking*.

Note: Of course, this idea also works for abstracting a set of abstract traces by a set of abstract states.

Abstracting a set of traces to a set of states

Define $\delta_\Sigma : \Sigma \to \mathit{Trace}(\tau^\Sigma)$ as $\delta_\Sigma(s) = \mathit{Trace}(\tau^\Sigma) \Downarrow s$.

Given state set, Σ, semantics τ_Σ, and traces $\mathit{Trace}(\tau^\Sigma)$, we apply the universal-abstraction construction and generate the Galois connection,

$$(\wp(\mathit{Trace}(\tau^\Sigma)), \supseteq)\langle \alpha^\forall, \gamma^\forall \rangle (\wp(\Sigma), \supseteq)$$

We have this concretization map, $\gamma^\forall : \wp(\Sigma) \to \wp(\mathit{Trace}(\tau^\Sigma))$:

$$\gamma^\forall(S) = \bigcup_{c \in S} \mathit{Trace}(\tau^A) \Downarrow c$$

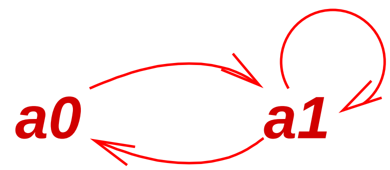
That is, $\gamma^\forall(S)$ builds all traces starting from states in S.

The abstraction map, $\alpha^\forall : \wp(\mathit{Trace}(\tau^\Sigma)) \to \wp(\Sigma)$:

$$\alpha^\forall(T) = \{c \in \Sigma \mid \mathit{Trace}(\tau^\Sigma) \Downarrow c \subseteq T\}$$

includes state $c \in \Sigma$ iff *all* possible traces starting from c are included in T — T “knows all about” c.

Example: $A = \{ a0, a1 \}$

$\tau^{A} =$ 

$\mathrm{Trace}(\tau^A) = A_0 \cup A_1,$ where $\begin{aligned} A_0 &= a_0 A_1 \\ A_1 &= a_1^\omega \cup a_1^+ A_0 \end{aligned}$

(Use the greatest-fixed point solution for A_0 and A_1.)

All the traces are infinite, and every trace has a suffix consisting of alternations of a_1 and a_0 or an infinite sequence of a_1s:

$$A_0 = \{a_0a_1a_1a_0..., a_0a_1a_0a_1a_0, ...a_0a_1a_1a_1..., \cdots\}$$

$$A_1 = \{a_1a_1a_0a_1..., a_1a_0a_1a_0a_1...., a_1a_1a_1..., \cdots\}$$

Some examples:

$$\begin{aligned}
\gamma^\forall(a_0) &= A_0 \\
\alpha^\forall(A_0 \cup \{a_1^\omega\}) &= \{a_0\} \\
\alpha^\forall(\gamma^\forall(a_0)) &= a_0 \\
\gamma^\forall\alpha^\forall(A_0 \cup \{a_1^\omega\}) &= A_0
\end{aligned}$$

Deriving a logic for traces abstracted to states

This abstraction of MiniLTL checks *trace* properties on the *states* that abstract the traces: for state set, A, $[\![\phi]\!]^\forall \subseteq A$ is defined as $[\![\phi]\!]^\forall = \alpha^\forall [\![\phi]\!]^A$, where $[\![\phi]\!]^A \subseteq \mathsf{Trace}(\tau^A)$.

A proof by induction shows that the above definition equals

$$[\![a]\!]^\forall = \alpha^\forall\{\kappa \in \mathsf{Trace}(A) \mid \kappa_0 \sqsubseteq a\}$$

$$[\![G\phi]\!]^\forall \; = \; \alpha^\forall \circ \mathsf{gen}^A \circ \gamma^\forall([\![\phi]\!]^\forall)$$

$$[\![F\phi]\!]^\forall = \alpha^\forall \circ \mathsf{fut}^A \circ \gamma^\forall([\![\phi]\!]^\forall)$$

Analysis of the embedded operations shows that

$$[\![a]\!]^\forall = \{a' \in A \mid a' \sqsubseteq a\}$$

$$[\![G\phi]\!]^\forall \; = \; \mathsf{gen}^\forall [\![\phi]\!]^\forall$$

$$\mathsf{gen}^\forall(M) = \{a \in A \mid \forall\kappa \in \mathsf{Trace}(\tau^A) \Downarrow a,\ \forall i \geq 0,\ \kappa_i \in M\}$$

$$[\![F\phi]\!]^\forall = \mathsf{fut}^\forall [\![\phi]\!]^\forall$$

$$\mathsf{fut}^\forall(M) = \{a \in A \mid \forall\kappa \in \mathsf{Trace}(\tau^A) \Downarrow a,\ \exists i \geq 0,\ \kappa_i \in M\}$$

We have:

$$[\![a]\!]^{\forall} = \{a' \in A \mid a' \sqsubseteq a\}$$

$$[\![G\phi]\!]^{\forall} \;=\; gen^{\forall}[\![\phi]\!]^{\forall}$$

$$gen^{\forall}(M) = \{a \in A \mid \forall \kappa \in Trace(\tau^{A}) \Downarrow a,\ \forall i \geq 0,\ \kappa_i \in M\}$$

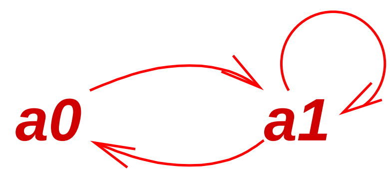
$$[\![F\phi]\!]^{\forall} = fut^{\forall}[\![\phi]\!]^{\forall}$$

$$fut^{\forall}(M) = \{a \in A \mid \forall \kappa \in Trace(\tau^{A}) \Downarrow a,\ \exists i \geq 0,\ \kappa_i \in M\}$$

We have just derived a fragment of the *branching-time logic* CTL, where F is CTL's AF and G is AG.

Example: ***A = { a0, a1 }***

$\tau^{A} =$



$a_0 \models^{\forall} F(at\ a1)$ – every trace from a_0 reaches a_1

$a_0 \models^{\forall} GF(at\ a1)$ – at every state reached by every trace from a_0, a_1 will be reached

$a_1 \not\models^{\forall} F(at\ a_0)$ – not all traces from a_1 reach a_0

We have: $$\begin{aligned}[\![G\phi]\!]^{\forall} &= gen^{\forall}[\![\phi]\!]^{\forall}\\ [\![F\phi]\!]^{\forall} &= fut^{\forall}[\![\phi]\!]^{\forall}\end{aligned}$$

Unfortunately, the definitions of $gen^{\forall}$ and $fut^{\forall}$ are defined on entire *traces* and not states! It would be more satisfying to have definitions stated in terms of states only.

When the state set, A, is finite, we can prove that the following definitions are equivalent to the ones seen earlier:

$$gen^{\forall}(M) = \bigcap_{i\geq 0} g_i, \quad \begin{aligned} g_0 &= A \\ g_{i+1} &= \{a \in A \mid a \in M \text{ and } \forall(a \to a'), a' \in g_i\}\end{aligned}$$

$$fut^{\forall}(M) = \bigcup_{i\geq 0} f_i, \quad \begin{aligned} f_0 &= \{\} \\ f_{i+1} &= \{a \in A \mid a \in M \text{ or } \exists(a \to a'), a' \in g_i\}\end{aligned}$$

These definitions calculate the states that can be reached by the transitions (*“branches”*), $a \to a'$, in τ^A. The definitions specify a model checker for *branching-time logic*.

The existential abstraction

The dual to the universal abstraction goes like this: We start with the same concretization map: $\gamma^{\exists} : \wp(A) \to \wp(\mathit{Trace}(\tau^A))$:

$$\gamma^{\exists}(S) = \bigcup_{a \in S} \mathit{Trace}(\tau^A) \Downarrow a$$

But we use this abstraction map: $\alpha^{\exists} : \wp(\mathit{Trace}(\tau^A)) \to \wp(A)$:

$$\alpha^{\exists}(T) = \{\kappa_0 \in A \mid \kappa \in T\}$$

This Galois connection defines an *existential abstraction*:

$$(\wp(\mathit{Trace}(\tau^A)), \subseteq)\langle \alpha^{\exists}, \gamma^{\exists} \rangle (\wp(A), \subseteq)$$

(Note the usual set inclusion for both powersets.)

We can use the existential abstraction to define a fragment of ECTL, $[\![\phi]\!]^{\exists}$, that possesses the EF and EG modalities. But the logic *overestimates* the traces that have property ϕ — it is a *possibly*-analysis rather than a necessarily analysis (like $[\![\phi]\!]^{\forall}$).

A dual Galois connection, based on a dual safety property, is needed to define a necessarily analysis that includes EF and EG. This is a topic that requires detailed development!

(See Dennis Dams's PhD thesis, Technical University Eindhoven, 1996.)

References

- A. Aho and J. Ullman. *Principles of Compiler Design*. Addison Wesley, 1977.
- B. Blanchet, et al. Design and implementation of a special purpose static analyzer for safety critical real-time embedded software. In The Essence of Computation, Springer LNCS 2566, 2002.
- P. Cousot and R. Cousot. Temporal Abstract Interpretation. POPL 1997.
- P. Cousot and R. Cousot. On abstraction in software verifi cation. Proc. CAV'02. Springer LNCS 2404, 2002.
- S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. Proc. CAV'97, Springer LNCS 1254, 1997.
- M. Müller Olm, et al. Model checking: a tutorial introduction. In Proc. 6th SAS, Springer LNCS 1694, 1999.
- H.R. Nielson, F. Nielson, and C. Hankin. *Principles of Program Analysis*, Springer, 1999.
- D. Schmidt. From trace sets to modal-transition systems by stepwise abstract interpretation. Workshop on structure-preserving relations, 2001. Available at `www.cis.ksu.edu/~schmidt/papers`