

Summary Program Analysis - Spring 2015 ETHZ

Flurin Rindisbacher

September 19, 2015

Contents

1	Abstract interpretation	4
1.1	Recipe for building a static analyzer	4
1.2	Partially Ordered Sets (posets)	4
1.3	Complete Lattices	5
1.4	Chains	5
1.5	Functions	5
1.5.1	Tarski's fixed point theorem	6
1.5.2	iterative algo for fixed point	6
1.6	Galois Connection	6
1.6.1	Galois Insertion	6
1.6.2	Key property of Galois connection	7
1.6.3	Some more properties	7
1.6.4	Composition	7
1.7	Function Approximation	7
1.7.1	general	7
1.7.2	Using a galois connection	7
1.7.3	Least precise approximation	8
1.7.4	Most precise approximation	8
1.7.5	Key Theorem 1: Least Fixed Point Approximation	8
1.7.6	Key Theorem 2: Least Fixed Point Approximation	8
2	Numerical domains I: Intervals	9
2.1	Abstract Interpretation Recipe by Example	9
2.2	Interval Domain	9
2.3	Cartesian Abstraction	9
2.4	Widening	10
2.5	Example	10
2.5.1	Concrete Domain	10
2.5.2	Collecting Semantics: F^c	11
2.5.3	Interval Abstraction	11
2.5.4	Arithmetic Expressions	11
2.5.5	Widening for Interval Domain	11
3	Numerical domains II: Octagons	12
3.1	Motivation	12
3.2	Basics	12
3.3	Encoding	15
3.4	Formalization	15

3.5	Closure(*)	15
3.6	Least Upper Bound (\sqcup_O)	15
3.7	Greatest Lower Bound (\sqcap_O)	15
3.8	Order (\sqsubseteq_O)	16
3.9	Widening (∇_O)	16
3.10	Transformers	16
3.10.1	Assignment	16
3.10.2	Conditional Statements	16
3.11	Example	16
4	Numerical domains III: Pentagons	17
4.1	General	17
4.2	Domain of Strict Upper Bounds (SUB)	17
4.3	Formalization	17
5	Applications: Analysis of HPC/GPU programs	20
5.1	Motivation	20
5.2	Goal	20
5.3	Conflict-free Checker	20
5.4	Unboundedness	22
5.4.1	Points-to Analysis	22
5.5	Flow-Insensitive Analysis	22
5.6	Computing abstract states	22
5.7	May Equal (Arrays) / Abstract Locations	25
5.8	Caveats	25
5.8.1	Domain-specific solution	25
5.9	Implementation	25
5.10	Limitations	25
5.11	Recap	25
6	Applications: Semantic Program Differencing	26
6.1	Motivation	26
6.1.1	Equivalence under abstraction	26
6.2	Their Approach	26
6.2.1	Finding the Analysis Order	27
6.2.2	Correlating Programs	27
7	SMT theories & Symbolic Execution	28
7.1	Intro	28
7.2	Decidability	28
7.3	TODO	30
7.4	Symbolic Execution	30
7.4.1	Existing Tools	30
7.4.2	Symbolic store	30
7.4.3	Semantics	30
7.4.4	Path Constraint	30
7.4.5	Limitation: Loops	30
7.4.6	Constraint solving	30
7.4.7	Concolic Execution - When constraint solving fails	31

8	Predicate Abstraction and Concurrency	32
8.1	Introduction	32
8.2	Theory	32
8.2.1	Domain	32
8.2.2	Concrete Transformers	32
8.3	Predicate Abstraction recipe	33
8.3.1	Abstract domain	34
8.3.2	Abstract Transformers	35
8.3.3	Iterate to a fixed point	35
8.3.4	Summary	36
8.4	Predicate Abstraction for Modern Concurrency	36
9	Synthesis from Examples	37
9.1	Problem Dimensions	37
9.2	Version Spaces	37
9.3	Version Space Algebra	38
9.3.1	Union	38
9.3.2	Join	38
9.3.3	Independent join	38
9.3.4	Transforms	38
9.4	Learning Version spaces	38
9.4.1	Inductive Definitions of Probabilities	39
9.5	Example: SMARTedit	40
9.5.1	Action	40
9.5.2	Version space	40
9.5.3	Learning	40
9.5.4	string searching version spaces	40
10	Dynamic race detection	41
10.1	Race conditions	41

Chapter 1

Abstract interpretation

A.I. - invented in late 70s by Patrick Cousot.

Frameworks for building analyzers: LLVM, Soot, WALA

1.1 Recipe for building a static analyzer

1. come up with an **abstract domain** (select based on the properties to prove)
2. define abstract semantics **for the programming language** w.r.t. to the abstract domain (define the **abstract transformer**, the effect of statement/expression on the abstract domain).
3. iterate until fixed point is reached

fixed point is an **over-approximation** of the program

1.2 Partially Ordered Sets (posets)

- Binary relation \sqsubseteq on a set L (subset of $L \times L$)
- **three properties**: reflexive, transitive, anti-symmetric
- Reflexive: $\forall p. p \sqsubseteq p$
- Transitive: $\forall p, q, r \in L. (p \sqsubseteq q \wedge q \sqsubseteq r) \implies p \sqsubseteq r$
- Anti-symmetric: $\forall p, q. (p \sqsubseteq q \wedge q \sqsubseteq p) \implies p = q$
- A poset (L, \sqsubseteq) is a set L equipped with a partial ordering \sqsubseteq .
- $p \sqsubseteq q$ intuitively means, that $p \implies q$
if $p \sqsubseteq q$ then p is "**more precise**" than q (p represents fewer concrete states than q)
- **Least** element (\perp) in Poset: $\forall p. \perp \sqsubseteq p$
Greatest element (\top) in Poset: $\forall p. p \sqsubseteq \top$
may not exist. but are unique if they do.

- $u \in L$ is an **upper bound** of $Y \subseteq L$ if $\forall p \in Y. p \sqsubseteq u$
- $l \in L$ is a **lower bound** of Y if $\forall p. l \sqsubseteq p$
- $\sqcup Y \in L$ is a **least upper bound** of Y if $\sqcup Y$ is an upper bound of Y and $\sqcup Y \sqsubseteq u$ whenever u is another upper bound of Y .
- $\sqcap Y \in L$ is **greatest lower bound** of Y if $\sqcap Y$ is a lower bound of Y and $\sqcap Y \sqsupseteq l$ whenever l is another lower bound of Y .
- $\sqcup Y$ and $\sqcap Y$ need not be in Y
- $p \sqcup q$ and $\sqcup\{p, q\}$ are the same

1.3 Complete Lattices

(L, \sqsubseteq, \sqcup) is a poset where $\sqcup Y$ and $\sqcap Y$ exist for any $Y \subseteq L$

1.4 Chains

Given a poset (L, \sqsubseteq) , a subset $Y \subseteq L$ is a **chain** if every two elements in Y are comparable (ordered)

$\forall x, y \in Y. (x \sqsubseteq y) \vee (y \sqsubseteq x)$ The **height** of a poset is the chain with the largest size.

1.5 Functions

Let $f : A \rightarrow B, g : A \rightarrow A, (A, \sqsubseteq)$ a poset and $x \in A$

- **monotone:**
 $\forall a, b \in A : a \sqsubseteq b \implies f(a) \leq f(b)$
 $\forall a, b \in A : a \sqsubseteq b \implies g(a) \sqsubseteq g(b)$
- **Fixed Points**
 x is fixed point iff $g(x) = x$
 x is a **post-fixedpoint** iff $g(x) \sqsubseteq x$
 $\text{Fix}(f)$ is the set of all fixed points
 $\text{Red}(f) =$ set of all post-fixedpoints
least fixed point ($\text{lfp} \sqsubseteq f \in L$): is fixed point and all other fixed points are above (\sqsubseteq)
a least fixed point may not exist
 When does a list fixed point exist? see Tarski's fixed point theorem 1.5.1
- **Function iterates:** $g^0(a), g^1(a), g^2(a)$ where $g^{n+1}(a) = g(g^n(a))$ we usually take $a = \perp$
- **completely meet-preserving:** for any subset $Y \subseteq A$: $f(\sqcap_2 Y) = \sqcap_1 \{f(y) | y \in Y\}$ (when \sqcap_1 and \sqcap_2 exists)
- **completely join-preserving:** for any subset $Y \subseteq A$: $f(\sqcup_1 Y) = \sqcup_2 \{f(y) | y \in Y\}$ (when \sqcup_1 and \sqcup_2 exists) Sheets 36,37 for examples
- join-preserving implies monotonicity

1.5.1 Tarski's fixed point theorem

if $(L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ is **complete lattice** and

$f : L \rightarrow L$ is a **monotone function**

then:

$lfp^{\sqsubseteq} f$ exists, and

$lfp^{\sqsubseteq} f = \sqcap Red(f) \in Fix(f)$

1.5.2 iterative algo for fixed point

Given a poset of finite height, a least element \perp and a monotone f :

The iterates $f^0(\perp), f^1(\perp), \dots$ form an increasing sequence which eventually stabilizes from some $n \in \mathbb{N}$.

$f^n(\perp) = f^{n+1}(\perp)$ and:

$$lfp^{\sqsubseteq} f = f^n(\perp) \quad (1.1)$$

Equation 1.1 leads to a simple iterative algorithm for computing $lfp^{\sqsubseteq} f$.

1.6 Galois Connection

Let $(L_1, \sqsubseteq_1, \sqcup_1, \sqcap_1)$ and let $(L_2, \sqsubseteq_2, \sqcup_2, \sqcap_2)$ be complete lattices.

Let $\alpha : L_1 \rightarrow L_2$ and let $\gamma : L_2 \rightarrow L_1$. Then α and γ form a **Galois connection** if:

- α is monotone
- γ is monotone
- $\alpha \circ \gamma$ is reductive: $\forall x_2 \in L_2 : \alpha(\gamma(x_2)) \sqsubseteq_2 x_2$
- $\gamma \circ \alpha$ is extensive: $\forall x_1 \in L_1 : \gamma(\alpha(x_1)) \sqsupseteq_1 x_1$

Equivalent definition (more useful for proofs):

$$\forall x_1 \in L_1, \forall x_2 \in L_2 : \alpha(x_1) \sqsubseteq_2 x_2 \iff x_1 \sqsubseteq_1 \gamma(x_2) \quad (1.2)$$

Intuition: α and γ are monotone because relationship between information and concrete is preserved in the abstract and vice versa.

Intuition 2: $\gamma \circ \alpha$ is extensive because α is indeed a correct approximation.

Lattice defined over L_1 captures the **concrete**. The other one captures the **abstract**. α is an **abstraction function** and γ is a **concretization function**. The galois connections links concrete and abstract.

1.6.1 Galois Insertion

Galois insertion is a Galois connection where γ is also injective. This means, that $\alpha \circ \gamma$ is the identity function and that α is also surjective.

Intuition: Each abstract element has a unique representation in the concrete.

1.6.2 Key property of Galois connection

The abstraction function α uniquely determines the concretization function γ and vice versa.

$$\begin{aligned} \underline{if} : (L_1, \sqsubseteq_1, \sqcup_1, \sqcap_1) &\xrightarrow[\alpha]{\gamma} (L_2, \sqsubseteq_2, \sqcup_2, \sqcap_2) \\ \underline{then} : \alpha(x) &= \sqcap_2 \{z \mid x \sqsubseteq_1 \gamma(z)\} \\ \gamma(z) &= \sqcup_1 \{x \mid \alpha(x) \sqsubseteq_2 z\} \end{aligned} \quad (1.3)$$

Before building the connection this way, need to check that α is completely join preserving or γ is completely meet preserving.

1.6.3 Some more properties

a galois connection implies, that α is completely join preserving and that γ is completely meet-preserving.

The other direction is not true. See counter example on page 40

1.6.4 Composition

Galois connections can be composed. E.g. use $\alpha_2 \circ \alpha_1$ to abstract from L_1 to L_3 via L_2 . Same for γ for concretisation.

1.7 Function Approximation

1.7.1 general

Given $f : A \rightarrow B$ and $g : A \rightarrow B$
 g approximates f if $\forall x \in A : f(x) \leq g(x)$

1.7.2 Using a galois connection

Given $(L_1, \sqsubseteq_1, \sqcup_1, \sqcap_1) \xrightarrow[\alpha]{\gamma} (L_2, \sqsubseteq_2, \sqcup_2, \sqcap_2)$
and

$f : L_1 \rightarrow L_1$

$g : L_2 \rightarrow L_2$

g approximates f if:

$\forall x \in L_1, \forall z \in L_2 : \alpha(x) \sqsubseteq_2 z \implies \alpha(f(x)) \sqsubseteq_2 g(z)$ Applying $f(x)$ and the abstracting stays below abstracting and then applying $g(z)$.

Equivalent statements for monotone functions:

- $\forall x \in L_1, \forall z \in L_2 : \alpha(x) \sqsubseteq_2 z \implies \alpha(f(x)) \sqsubseteq_2 g(z)$
- $\forall z \in L_2 : g(z) \sqsupseteq_2 \alpha(f(\gamma(z)))$
- $\forall x \in L_1 : \alpha(f(x)) \sqsubseteq_2 g(\alpha(x))$
- $\forall z \in L_2 : f(\gamma(z)) \sqsubseteq_1 \gamma(g(z))$

1.7.3 Least precise approximation

$g(z) = \top$ is a sound but imprecise approximation. Existence of $g(z)$ is guaranteed.

Soundness: $\forall z \in L_2 : \top \sqsupseteq_2 \alpha(f(\gamma(z)))$

1.7.4 Most precise approximation

- a composition of monotone functions is monotone
- $g(z) = \alpha(f(\gamma(z)))$ is the **best abstract function** that satisfies the condition $\forall z \in L_2 : g(z) \sqsupseteq_2 \alpha(f(\gamma(z)))$
- hard to implement such a $g(z)$ algorithmically
- can come up with $g(z)$ that has same behaviour as $\alpha(f(\gamma(z)))$ but different implementation. Any such $g(z)$ is referred to as the **best transformer**.

1.7.5 Key Theorem 1: Least Fixed Point Approximation

if we have

1. monotone functions $F : C \rightarrow C$ and $F^\# : A \rightarrow A$
2. $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ forming a Galois Connection
3. $\forall z \in A : \alpha(F(\gamma(z))) \sqsubseteq_A F^\#(z)$ (that is, $F^\#$ approximates F)

then:

$$\alpha(lfp(F)) \sqsubseteq_A lfp(F^\#) \quad (1.4)$$

Important because it goes from local function approximation to global. A key theorem in program analysis. The premises are usually proved manually. Then a least fixed point can be automatically computed and the result is sound.

1.7.6 Key Theorem 2: Least Fixed Point Approximation

If α and γ do not form a Galois connection (e.g because α is not monotone) then use the following definition of approximation:

$$\forall z \in A : F(\gamma(z)) \sqsubseteq_c \gamma(F^\#(z)) \quad (1.5)$$

This brings us to the second key theorem:

If we have:

1. monotone functions $F : C \rightarrow C$ and $F^\# : A \rightarrow A$
2. $\gamma : A \rightarrow C$ is monotone
3. $\forall z \in A : F(\gamma(z)) \sqsubseteq_c \gamma(F^\#(z))$ (that is, $F^\#$ approximates F)

then:

$$lfp(F) \sqsubseteq_c \gamma(lfp(F^\#)) \quad (1.6)$$

Chapter 2

Numerical domains I: Intervals

2.1 Abstract Interpretation Recipe by Example

0. Since these steps depend on the concrete function it makes sense to define the function in the concrete once and for all. Then the three step recipe can be applied.
1. Abstract domain: Interval domain.
The interval domain is a complete lattice with \perp at the bottom and $[-\infty, \infty]$ at top. Other Elements are for example $[0, 0]$, $[1, 1]$, $[-2, 2]$ etc.
2. Abstract transformers:

2.2 Interval Domain

Let the interval domain be: $(L^i, \sqsubseteq_i, \sqcup_i, \sqcap_i, \perp_i, [-\infty, \infty])$

We denote $Z^\infty = Z \cup \{-\infty, \infty\}$

The set $L^i = \{[x, y] | x, y \in Z^\infty, x \leq y\} \cup \{\perp_i\}$

For a set $S \subseteq Z^\infty$, $\min(S)$ returns the minimum number in S, $\max(S)$ returns the maximum number in S.

- $[a, b] \sqsubseteq_i [c, d]$ if $c \leq a$ and $b \leq d$
- $[a, b] \sqcup_i [c, d] = [\min(\{a, c\}), \max(\{b, d\})]$
- $[a, b] \sqcap_i [c, d] = [\text{meet}(\max(\{a, c\}), \min(\{b, d\}))]$
where $\text{meet}(a, b)$ returns $[a, b]$ if $a \leq b$ and \perp_i otherwise.

2.3 Cartesian Abstraction

Useful for abstracting the set of states reachable at a program point. Cartesian abstraction is used when we are interested in properties that do not involve relationships between variables.

Interpretation: $a^x(\{x \mapsto 5, y \mapsto 7\}, \{x \mapsto 8, y \mapsto 9\}) = 1 \rightarrow \{x \rightarrow \{5, 8\}, y \rightarrow$

$\{7, 9\}$ So we lose the correlation between variables. Then applying $\gamma^x(1 \rightarrow \dots)$ will result in all possible combinations of x and y . (Cartesian Abstraction)?

TODO: try to understand this. (Page 20).

Theorem 1 *If (L, \sqsubseteq, \sqcup) is a complete lattice, and X is a set, then the set of functions, from X into L is a complete lattice $(X \rightarrow L, \sqsubseteq, \sqcup)$*

2.4 Widening

When a variable keeps increasing the iterates $F^i(\perp)$ will keep going and it's not possible to compute all the iterates to find a fixed point. Solution: **Widening** (approximate the fixed point).

$\nabla : L \times L \rightarrow L$ is called a widening operator if:

- $\forall a, b \in L : a \sqcup b \sqsubseteq a \nabla b$
- if $x^0 \sqsubseteq x^1 \sqsubseteq x^2 \sqsubseteq \dots \sqsubseteq x^n \sqsubseteq \dots$ is an increasing sequence then $y^0 \sqsubseteq y^1 \sqsubseteq y^2 \sqsubseteq \dots \sqsubseteq y^n$ stabilizes after a finite number of steps.
where $y^0 = x^0$ and $\forall i \geq 0 : y^{i+1} = y^i \nabla x^{i+1}$

Widening is completely independent of the Function F . Much like join it is an operator defined for the particular domain.

Theorem 2 *If L is a complete lattice, $\nabla : L \times L \rightarrow L, F : L \rightarrow L$ is monotone then the sequence*

$$y^0 = \perp$$

$$y^1 = y^0 \nabla F(y^0)$$

$$y^2 = y^1 \nabla F(y^1)$$

...

$$y^n = y^{n-1} \nabla F(y^{n-1})$$

will stabilize after a finite number of steps with y^n being a post-fixedpoint of F . From Tarskis Theorem 1.5.1 we know that a post-fixedpoint is above the least fixed point.

2.5 Example

This example works on the interval domain. Every variable is a number.

2.5.1 Concrete Domain

At every program counter (label, line-number) we want to keep the set of concrete states arising at that label.

$$1 \rightarrow \{\sigma_3\}$$

$2 \rightarrow \{\sigma_4, \sigma_5\}$ **Intuition:** For a given program, just take all reachable states at a given label.

Each element in the domain is a map: $\lambda l. \{ \}$ e.g. $1 \rightarrow \{\sigma_3\}$

2.5.2 Collecting Semantics: F^c

For every label we store the join (\cup) of $[action]_c(m(l'))$. **See sheet 15.**

$(l', action, l)$ must exist for every transition in the program. E.g $(1, i < 10, 2)$ and $(1, i \geq 10, 5)$ would be used for a program where on line 1 an if-condition "if($i < 10$)" starts and on line 5 the else-clause continues.

If we have $(l', action, l)$ for every possible transition then we are sound. If we have $(l', action, l)$ even for transitions that do not occur in the program then we are imprecise. We unnecessarily create more flows.

Possible actions are in our case: Boolean expressions, assignments and skip.

2.5.3 Interval Abstraction

Page 22:

$\alpha^i(C)(l)x = [\min(C(l)x), \max(C(l)x)]$ if $C(l)x \neq \emptyset$ (\perp otherwise)

$\gamma^i(M)(l)x = v \in Z \mid l \leq v \wedge v \leq u$ if $M(l)x = [l, u]$ (\perp otherwise)

2.5.4 Arithmetic Expressions

Adding \perp to anything else yields \perp .

Otherwise $[x, y] + [z, q] = [x + z, y + q]$ **TODO:** check page 33 $pair_{\leq}()$ definition.

2.5.5 Widening for Interval Domain

$[a, b] \nabla_i [c, d] = [e, f]$ where:

if $c < a$, then $e = -\infty$, else $e = a$

if $d > b$, then $f = \infty$, else $f = b$

If one of the operands is \perp the result is the other operand.

Intuition: If endpoint is unstable, move its value to the extreme case.

Chapter 3

Numerical domains II: Octagons

3.1 Motivation

- Interval domain only captures bounds of program variables
- Many verification tasks require relational invariants
- E.g. Buffer overflow, Concurrency bugs, Aliasing
- Tradeoff between complexity and expressive power
- Asymptotic Complexity: Interval \ll Pentagon \ll Octagon \ll Polyhedra
- Expressive Power: Interval \ll Pentagon \ll Octagon \ll Polyhedra

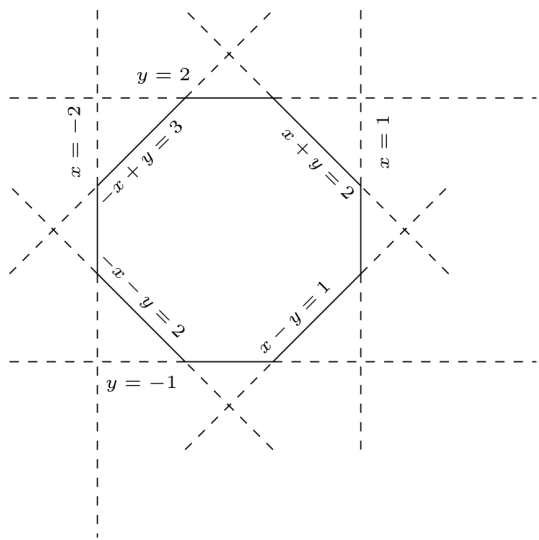
3.2 Basics

- Cubic time complexity.
- Used to verify correctness of software in Airbus.
- supports limited set of linear inequalities between program variables
- Inequalities
 - Binary: $\pm v_j \pm v_i \leq c, v_i \neq v_j$
 - Unary: $\pm v_i \leq d$
 - $c, d \in \mathbb{R} \cup \{\infty\}$
 - If an inequality does not exist, then its bound is ∞
 - inequalities limit the set of possible values taken by program variable
 - for n variables there are $2n^2$ possible inequalities
- octagon can be represented by more than one set of inequalities (e.g. $(x \leq 4) \wedge (y \leq 6)$ or $(x \leq 4) \wedge (y \leq 6) \wedge (x + y \leq 10)$)

- galois insertions need a unique representation of octagons
- the set with maximum inequalities representing an octagon is unique. See Closure 3.5

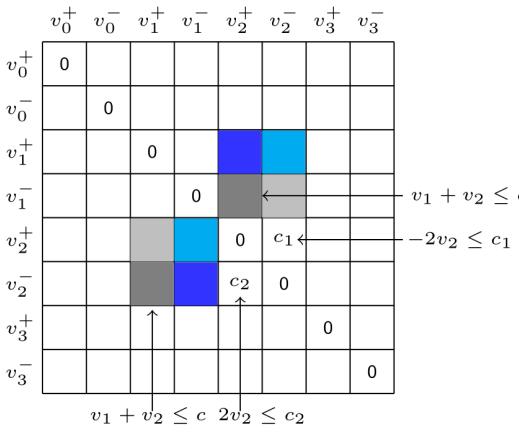
Octagons

- Let L be the set of Octagonal inequalities between n variables, then an octagon is conjunction of all the inequalities in set L



Encoding of Octagons

- Each octagonal inequality can be encoded as an element of a matrix m
- Each variable v_i is unfolded into $v'_{2i} = v_i^+$ and $v'_{2i+1} = v_i^-$
- $m_{i,j} = c$ represents $v'_j - v'_i \leq c$
- $v_i + v_j \leq c$ can be represented as:
 $v_j^+ - v_i^- \leq c$ and
 $v_i^+ - v_j^- \leq c$



3.3 Encoding

Matrix m encodes the inequalities. For k Variables m has size $2k \times 2k$. Each v_i is unfolded into $v'_{2i} = v_i^+$ and $v'_{2i+1} = v_i^-$.
 $m_{i,j} = c$ represents $v'_j - v'_i \leq c$
 $v_i + v_j \leq c$ can be represented as

- $v'_j - v_i^- \leq c$
or
- $v_i^+ - v'_j \leq c$

$diag(M) = 0$ obviously.

3.4 Formalization

- The Octagon domain: $(O^O, \sqsubseteq_O, \sqcup_O, \sqcap_O, \perp_O, \top_O)$
- \perp_O represents bottom element that contains an unsatisfiable set of inequalities.
- O is the set of all octagons
- $O^O = O \cup \{\perp_O\}$
- T_O top element for which the bound for all inequalities is ∞

3.5 Closure(*)

The closure operator produces a unique octagon representation. Binary inequalities such as $v_i - v_j \leq c_1$ and $v_j - v_k \leq c_2$ are comined to obtain $v_i - v_k \leq c_1 + c_2$. If the octagon already contains $v_i - v_k \leq c$ then keep $v_i - v_k \leq \min(c, c_1 + c_2)$. (This is same as applying Floyd Warshall on the octagon matrix). The set produced by this is maximal and unique.

3.6 Least Upper Bound (\sqcup_O)

Union of two octagons is not necessarily an octagon. \sqcup is therefore an approximation.

First apply closure to both operands. Then compute union by taking piecewise maximum of bounds of corresponding inequalities.

$$(x \leq 5) \wedge (x + y \leq 10) \sqcup_O (x \leq 4) \wedge (x + y \leq 11) = (x \leq 5) \wedge (x + y \leq 11)$$

3.7 Geatest Lower Bound (\sqcap_O)

Intersection of two octagons is always an octagon. Take piecewise minimum of bounds to calculate \sqcap_O .

3.8 Order (\sqsubseteq_O)

An octagon O_1 is included inside another octagon O_2 iff the bounds of each inequality in O_1 is \leq than the corresponding inequality in O_2 . Inclusion relation is used for octagon ordering. The closed octagon is the smallest octagon as per \sqsubseteq_O among the set of octagons abstracting the same concrete values.

3.9 Widening (∇_O)

Widening requires the first operand to not be closed. Widening increases the number of inequalities with ∞ whereas closure does the reverse. To widen just set the bound to ∞ if it increases.

3.10 Transformers

3.10.1 Assignment

Transformer is precise for octagonal assignments but only approximate for those non octagonal.

Octagonal assignments:

- $x = c$
Add inequalities $(x \leq c)$ and $(-x \leq c)$ to the octagon and close it.
- $x = x + c$
Subtract c from inequalities having negative coefficient for x . Add c to inequalities having positive coefficient for x . The result is already closed.
- $x = y + c$
Add inequalities $(x - y \leq c)$ and $(y - x \leq c)$ to the octagon and close it.

Non-octagonal Assignments: $x_j = e$ where $x_j - e$ is non octagonal.

1. Compute bounds $[a, b]$ for e using interval arithmetic
E.g $e = [a_0, b_0] + \sum_{i=1}^n [a_i, b_i]x_i$ where each x_i has bounds $[c_i, d_i]$ then $[a, b] = [a_0, b_0] + \sum_{i=1}^n [a_i, b_i] \times [c_i, d_i]$
2. Add constraints of the form $\pm x_i \pm x_j \leq [a, b] \pm [c_i, d_i]$ to the octagon.
3. close the octagon

3.10.2 Conditional Statements

Conditional statements encode constraints which can be added to the input octagon. There are octagonal and non octagonal constraints. Similar to assignment octagonal constraints are precise whereas the other constraints are approximated.

3.11 Example

See page 24 1

Chapter 4

Numerical domains III: Pentagons

4.1 General

- Pentagon domain is a reduced product of two domains: Interval ($c \leq x \leq d$) and Strict Upper Bounds (SUB) ($x < y$)
- Quadratic space complexity
- Quadratic time complexity
- Useful for checking array out of bounds error
 - interval component takes care of index underflow ($index \geq 0$)
 - the SUB component takes care of index overflow ($index < array.length$)

4.2 Domain of Strict Upper Bounds (SUB)

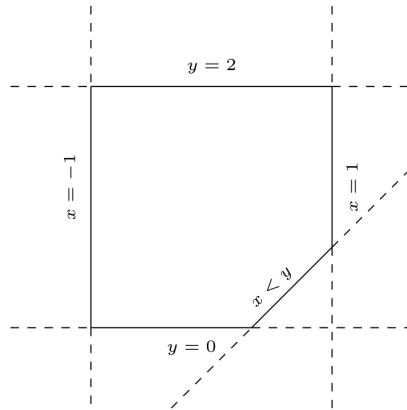
- for each variable x store the list $s(x)$ of all other variables y s.t. $x < y$
- SUB domain: $\{S^S, \sqsubseteq_S, \sqcup_S, \sqcap_S, \perp_S, \top_S\}$
- $\perp_S \iff \exists x, y$ s.t., $y \in s(x) \wedge x \in s(y)$
- S is the set of all SUB inequalities, $S^S = S \cup \{\perp_S\}$
- $\top_S \iff \forall x, s(x) = \emptyset$

4.3 Formalization

- $s_1 \subseteq_S s_2 \iff \forall x, s_1(x) \supseteq s_2(x)$
- $s_1 \sqcup_S s_2 = \forall x. s_1(x) \cap s_2(x)$
- $s_1 \sqcap_S s_2 = \forall x. s_1(x) \cup s_2(x)$
- $s_1 \nabla_S s_2 = \forall x. s_1(x) \subseteq s_2(x) ? s_2(x) : \emptyset$

- Closure is not performed to avoid cubic complexity.
 - therefore no galois insertion
 - Domain loses precision for various operators

- Let i be the set of interval and s be the set of SUB constraints over n variables, then a Pentagon is the conjunction of all the constraints in i and s represented as a tuple (i,s)



- The Pentagon Domain: $\{P^p, \sqsubseteq_p, \sqcup_p, \sqcap_p, \perp_p, \top_p\}$
- $(i, s) = \perp_p \iff (i = \perp_i) \vee (s = \perp_s)$
- P is the set of all Pentagons, $P^p = P \cup \{\perp_p\}$
- $(i, s) = \top_p \iff (i = \top_i) \wedge (s = \top_s)$

Pentagon Abstract Domain

- $(i_1, s_1) \sqsubseteq_p (i_2, s_2) \iff (i_1 \sqsubseteq_i i_2) \wedge (\forall x, \forall y \in s_2(x) \text{ s.t. } y \in s_1(x) \vee \sup(i_1(x)) \leq \inf(i_2(x)))$
- $(i_1, s_1) \sqcup_p (i_2, s_2) = (i_1 \sqcup_i i_2, s' \cup s'' \cup s''')$, where
 - $s' = \forall x. s_1(x) \cap s_2(x)$
 - $s'' = \forall x. \{y \in s_1(x) \mid \sup(i_2(x)) < \inf(i_2(y))\}$
 - $s''' = \forall x. \{y \in s_2(x) \mid \sup(i_1(x)) < \inf(i_1(y))\}$
- $(i_1, s_1) \sqcap_p (i_2, s_2) = (i_1 \sqcap_i i_2, s_1 \sqcap_s s_2)$
- $(i_1, s_1) \nabla_p (i_2, s_2) = (i_1 \nabla_i i_2, s_1 \nabla_s s_2)$

Chapter 5

Applications: Analysis of HPC/GPU programs

5.1 Motivation

Due to FJ, Cilk, X10, DPJ, TPL, CUDA there's renewed interest in structured parallel languages. Determinism is wanted in applications that use such languages. There are lots of parallel algorithms: Scientific Computing, Signal Processing, Encryption, Sorting, Searching, String Indexing

Determinism: for the same input, produce the same output.

5.2 Goal

Prove determinism. Any pair of terminating executions starting with equivalent input states, end in equivalent output states. Proving arbitrary programs deterministic is hard, instead, we prove a stronger property which implies determinism:

Conflict-Freedom meaning, parallel threads always access disjoint memory.

1. Compute all reachable concrete states
2. Check if each state is conflict-free

We denote conflict as a state where 2 threads are enabled to access the same memory location and one of these accesses is a write. **Checking Conflict-Freedom is potentially unbounded.**

5.3 Conflict-free Checker


Conflict-Free Checker

For every program state σ

For all pairs of threads t_1 and t_2 in σ :

if t_1 and t_2 are **enabled** 

temporal check

if $\text{Loc}(t_1, \sigma) = \text{Loc}(t_2, \sigma)$ 

spatial check

exit ("Program may be **non-deterministic**")

exit ("Program is **deterministic**")

$\text{Loc}(t, \sigma)$ returns the memory location accessed by transition about to be accessed by thread t

5.4 Unboundedness

Heap, range of array indices and number of threads is unbounded.

Dealing with Unboundedness

- Unbounded Heap
 - Compute finite set of abstract locations
 - Using flow-insensitive **points-to analysis**
- Unbounded range of array indices
 - Compute symbolic index constraints
 - Using **numerical abstractions**
- Unbounded number of threads (not discussed in class)

5.4.1 Points-to Analysis

Terms

Two pointers p and q are aliases if they point to the same memory location. (p, A) is a points-to pair where p holds the address of object A . For two points-to pairs $(p, A), (r, A)$ p and r are aliases.

Allocation Sites

Heap is divided into a fixed partitions. All objects allocated at the same program point are represented by a single "abstract object".

5.5 Flow-Insensitive Analysis

Just ignore if conditions and look at every statement for points-to analysis. E.g

```
p := new Array 5; // allocation site A1
q := new Array 5; // allocation site A2
if p=q then
  z := p
else
  z := q
```

will output $points - to(z) = A1, A2$.

5.6 Computing abstract states

Using sequential analysis (based on numerical analysis and pointer analysis) we compute for each thread. **see Page 34**. First we build for each label (program line) the abstract states. E.g. $\sigma_1 = \{pc = 1, idx = 2 * tid - ps\}$ for a first program line looking like:

```

void update(double[] [] G, int start, int last, double c1, double c2, int nm, int ps){
    for(tid=start; tid<last; tid+=1){
1:      double [] Gi = G[i];
    ...
    }
}

```

For each thread id (*tid*) we can build cartesian states. E.g for *tid* = 1 and label 1 ($pc_1 = 1$):

$pc_1 = 1, ps = 0, idx_1 = 2$

$pc_1 = 1, ps = 1, idx_1 = 1$

$pc_1 = 1, ps = 2, idx_1 = 0$

And for *tid* = 4:

$pc_4 = 1, ps = 0, idx_4 = 8$

$pc_4 = 1, ps = 3, idx_4 = 5$

$pc_4 = 1, ps = 7, idx_4 = 1$

Combining cartesian states for different threads will give us the program states:

$pc_1 = 1, idx_1 = 2, pc_4 = 1, idx_4 = 8, ps = 0$

Since *ps* is a parameter to the function combining two cartesian states with different *ps* values does not make sense. But it's OK to combine two threads but on different program lines ($pc_i = k, pc_j = k', k \neq k', i \neq j$).

Summary:

- compute invariants for each thread. denote computed values using an expression
- build cartesian states. "fill" in the missing values to compute variables (many possible combinations).
- combine abstract states for different threads.

Recall the conflict free checker: 5.3. Using the abstract states we can now define an abstract conflict-free checker:

Abstract Conflict-Free Checker

For every pair of **abstract** states σ_i, σ_k :
if st_i in σ_i and st_k in σ_k **may happen** in parallel
A = combine σ_i, σ_k and $(tid_i \neq tid_k)$
if $Aloc(st_i)$ and $Aloc(st_k)$ **may equal** in A
exit ("Program may be **non-deterministic**")
exit ("Program is **deterministic**")

temporal check

spatial check

- Combine means meet in the domain
- Note: thread constraint $(tid_i \neq tid_k)$ not expressible in polyhedra)

A may happen analysis can be used for the may happen parts. There are many works on this. **TODO?**

5.7 May Equal (Arrays) / Abstract Locations

$ALoc(G[i] = 128) := (G, i)$ with G a pointer- and i and integer variable. When may two $ALoc(a, i)$ and $ALoc(b, k)$ equal? When $(a \text{ may alias } b) \wedge (i \text{ may overlap } k)$. With $\text{may alias} = \text{points} - \text{to}(a) \cap \text{points} - \text{to}(b) \neq \emptyset$ and $\text{may overlap} = A_N \sqcap (i = k) \neq \perp$ **TODO: not sure what A_n is. page 49**

5.8 Caveats

Heap abstraction is imprecise.

- Aliasing information inside reference arrays is lost. Example:

```
Object A[], double G[][];
```
- So, points-to information is not enough
- We want $\forall i, j, A : i \neq j \implies A[i] \neq A[j]$ but this is very difficult to prove in general.

5.8.1 Domain-specific solution

Most numerical HPC/GPU programs initialize reference arrays only with fresh objects $A[i] = \text{new Object}()$; and never update afterwards. Easy to prove as a global invariant.

5.9 Implementation

- Soot (Analysis works on Jimple intermediate representation)
- Apron library (for numerical invariants)
- Benchmarked using Java JGF benchmarks

5.10 Limitations

- cannot handle some kinds of non-linear constraints $(A[x*N + z] = c$ where N never changes
- atomic sections
- Accesses from nested primitive arrays: $A[B[i]] = 5$ where $B[i]$'s are distinct

5.11 Recap

To automatically prove determinism prove a stronger property: conflict-freedom.

Chapter 6

Applications: Semantic Program Differencing

6.1 Motivation

Do two programs behave the same. Has an example *print_numbers_v_6_9()* and *print_numbers_v_6_10()* with syntactic differences but the same output. Are they the same? When running the programs and comparing the difference you can only spot cases where procedures differ. You cannot prove equivalence for most programs. **Abstract Semantic Differencing** FTW! Either prove equivalence or characterize their difference (find differing program states that come from the same input).

Sound: Equivalence guaranteed Precise: Report few differences

6.1.1 Equivalence under abstraction

Equivalence under abstraction does not entail equivalence between the concrete values it represents.

6.2 Their Approach

Nimrod Partush, Eran Yahav, Technion, Israel

- Analyze P and P' together
 - define correlating semantics that interprets the programs together
 - Interpret in some ordering of their steps
 - Abstract the correlating semantics (to handle infinite-state programs)
- Search for the program ordering that allows the abstraction to best capture equivalence.

Idea: join (\sqcup) states only if they hold equivalence for the same variables.

$\{x < 0, res = res' = -1\} \sqcup \{x > 0, res = res'\} = \{x = \top, res = res' = [-1, 1]\}$

6.2.1 Finding the Analysis Order

Sequential order has problems: When one programs analysis has finished its values have been abstracted and equivalence is lost. Imagine a $M = n \times m$ Matrix here. $m_{1,1}$ is the abstract state where both programs are on line 1. How do we check the states?

- sequential $m_{1-n,1}$ then $m_{n,1-m}$:
fails to see that two identical programs are equivalent (they never intersect on a state).
- Lock-step $m_{1,1}, m_{2,1}, m_{2,2}, m_{3,2}$:
fails for programs with different number of lines
- all possible interleavings:
need to check every element of M (aka every possible interleaving)

6.2.2 Correlating Programs

Previous work (SAS 13')

Previously they joined the programs into one program which holds both program semantics. Assumed syntactic similarity and used a syntactic diff. Transformation tool *ccc* is available on github.

Problem: when syntactic difference is vast, the composition will be sequential. Leads to imprecise (and useless) results.

Speculative Exploration

Using a speculation window k both programs are explored k steps distributed over both programs.

- 0 over P and k over P'
- 1 over P and $k - 1$ over P'
- etc.

TODO: read again.

Chapter 7

SMT theories & Symbolic Execution

7.1 Intro

Validity in first order logic (FOL) is undecidable, while validity in particular first order theories is (sometimes) decidable. These theories allow us to capture structures which are used by programs (arrays, ints etc.) and enable reasoning about them.

Formulas in each theory are constructed with a specific set of function, predicate and constant symbols. This is the signature of the theory called Σ . Using Σ , logical connectives (\wedge, \rightarrow) and quantifiers first order formulas can be built. Each theory comes with a set of axioms (FOL formulas) called A , which only contain elements from the signature. A formula F in the theory is valid if all interpretations that satisfy the axioms in A also satisfy the formula. Some theories are meant to be used with a particular interpretation (e.g. in theory of integers formulas are interpreted over ints). A fragment of a theory consists of a subset of the possible formulas expressible in the theory. A theory is decidable if for every formula in the theory we can automatically check whether the formula is valid or not. Similarly for fragments of a theory.

7.2 Decidability

Decidability is mainly needed to achieve 100% automation. If the theory is not decidable sometimes the theorem prover (e.g. Z3, Yices) which is used may succeed.

SMT theories: Decidability

Theory	Description	Full Fragment	No Quantifiers
T_E	Equality	NO	YES
T_{PA}	Peano arithmetic	NO	NO
T_N	Presburger arithmetic	YES	YES
T_Z	Linear Integers	YES	YES
T_R	Reals (with *)	YES	YES
T_Q	Rationals (without *)	YES	YES
T_{RDS}	Recursive Data Structures	NO	YES
T_{RDS}^+	Acyclic Recursive Data Structures	YES	YES
T_A	Arrays	NO	YES
$T_A^=$	Arrays with extensionality	NO	YES

(source: The Calculus of Computation, Manna and Bradley)

7.3 TODO

pages 24 with all those theories.

7.4 Symbolic Execution

Widely used in practice. Symbolic Execution keeps two formulas at any point during program execution: **symbolic store** and a **path constraint**. The symbolic state is the conjunction of these two formulas.

7.4.1 Existing Tools

Stanford's KLEE, NASA's Java PathFinder, Microsoft Research's SAFE, UC Berkeley's CUTE, EPFL's S2E.

7.4.2 Symbolic store

$\sigma_s : Var \rightarrow Sym$ maps each variable to a value at any given time. Example:
 $\sigma_s : x \mapsto x0, y \mapsto y0$

7.4.3 Semantics

Arithmetic expression evaluation simply manipulates the symbolic values. Let $\sigma_s : x \mapsto x0, y \mapsto y0$. Then, $z = x + y$ will produce the symbolic store: $x \mapsto x0, y \mapsto y0, z \mapsto x0 + y0$

7.4.4 Path Constraint

Records all branches taken so far. Typically a decidable logical fragment without quantifiers. Is true at the start of the analysis. Evaluation of conditionals affects the path constraint but not the symbolic store.

7.4.5 Limitation: Loops

Unbounded loops run forever in symbolic execution. Easiest solution: provide some loop bound meaning under-approximate (under-approximation: only feasible paths? **TODO**). Another solution is to provide loop invariants. But this is rarely used for large programs because it's difficult to provide such invariants manually and it can lead to over-approximation. Static analysis can infer loop variants.

7.4.6 Constraint solving

It's important that

1. a SMT solver supports as many decidable logical fragments as possible (some tools even use more than one SMT solver)
2. a SMT solver can solve large formulas quickly
3. the symbolic execution engine tries to reduce the burden in calling the SMT solver by exploring domain specific insights.

A key-optimization is obviously caching. The symbolic execution engine can keep a map (cache) of formulas to a satisfying assignment for the formula. For new formulas it can the first access the cache before calling the SMT solver.

7.4.7 Concolic Execution - When constraint solving fails

SMT solvers do not handle non-linear constraints well (e.g $z = y*y$, ... page 47). Concolic Execution combines both symbolic execution and concrete (normal) execution. The program runs as usual (with some input that needs to be given) and also maintains the symbolic information. For example when there's a *read()* use two stores. one with actual values (e.g. 22,1, etc.) and one with the symbolic information. Run several times. If for example the then branch is reached and we need the else brnach. negate path constraint and run again. See examples on page 49.

Chapter 8

Predicate Abstraction and Concurrency

8.1 Introduction

Predicate Abstraction is used to automatically verify: C device drivers (SLAM project at MSR), distributes concurrent algorithms, biological systems (PRISM at Oxford), security properties.

8.2 Theory

8.2.1 Domain

Concrete Domain

$Lab \rightarrow p(Store)$ **TODO: powerset symbol?**

e.g. $2 \rightarrow \{\{x = 1, y = 3\}, \{x = 2, y = 2\}\}$

Logical Domain

Capture the set of stores with a FOL formula. Example:

$1 \rightarrow x = 1 \wedge y = 2$

$2 \rightarrow (x = 1 \wedge y = 3) \vee (x = 2 \wedge y = 2)$ Then a Galois Connection between $Lab \rightarrow p(Store)$ and $Lab \rightarrow FOL$ can be setup.

8.2.2 Concrete Transformers

$F^{FOL} : (Lab \rightarrow FOL) \rightarrow (Lab \rightarrow FOL)$

$$F^{FOL}(m)l = \begin{cases} true & \text{if } l \text{ is initial label} \\ \bigvee_{(l', action, l)} \llbracket action \rrbracket_{FOL}(m(l')) & \text{otherwise} \end{cases}$$

Effect of action on a FOL formula

$\llbracket action \rrbracket_{FOL} : FOL \rightarrow FOL$

Examples:

$$\begin{aligned}
\llbracket \text{skip} \rrbracket_{FOL}(x = 1 \wedge y = 2) &= (x = 1 \wedge y = 2) \\
\llbracket (x > 5) \rrbracket_{FOL}(x > 0 \wedge y = 2) &= (x > 5 \wedge y = 2) \\
\llbracket (x := 7) \rrbracket_{FOL}((x < 0) \wedge (x = 5)) &= (x = 7)
\end{aligned}$$

Strongest post condition

The best transformer for $\llbracket \text{action} \rrbracket_{FOL}$ is the strongest post-condition (sp).

$$\llbracket \text{action} \rrbracket_{FOL}(\psi) = sp(\psi, \text{action})$$

With $sp(\psi, \text{action}) = \psi'$ The formula ψ' is the most precise (strongest) set of successor stores of ψ as determined by action.

$$sp(\psi, \text{skip}) = \psi$$

$$sp(\psi, b) = \psi \wedge b$$

$$sp(\psi, x := a) = \exists v : x = a[v/x] \wedge \psi[v/x]$$

Quantifiers are difficult to handle for automated reasoning engines and many decidable fragments do not allow quantifiers.

Weakest Pre-Condition

$$wp(\psi, \text{skip}) = \psi$$

$$wp(\psi, b) = b \Rightarrow \psi$$

$$wp(\psi, x := a) = \psi[a/x]$$

No quantifiers!

linking sp with wp

Theorem 3 $sp(\psi, \text{action}) \Rightarrow \psi' \equiv \psi \Rightarrow wp(\psi', \text{action})$

Using weakest preconditions we get rid of quantifier. **TODO:** check again.

Undecidability

FOL is undecidable. If ψ in FOL is not valid, there is no algorithm guaranteed to terminate. Can't guarantee termination of iterating F^{FOL} .

8.3 Predicate Abstraction recipe

1. Come up with an abstract domain (based on the type of properties you want to prove)
2. Define abstract semantics for the programming language w.r.t. the abstract domain from step 1.
 - define the abstract transformers (effect of statement on the domain)
 - prove that the abstract semantics are sound w.r.t concrete semantics of the programming language
3. iterate abstract transformers over the abstract domain until fixed point

8.3.1 Abstract domain

We define F to be a finite set of predicates $F = \{\phi_1, \phi_2, \dots, \phi_n\}$ where the predicates are selected from a decidable logical fragment. Examples for the predicates are:

$$\phi_1 := x > 0 \wedge x = (x + 1)$$

$$\phi_2 := z + y > 5 \wedge x = (y + y) \text{ (over theory of integers).}$$

Using the predicates we define a cube which is a conjunction of predicates and negated predicates where each ϕ_i appears at most once. Cubes can be represented as vectors of size n where the value at index i indicates if ϕ_i is negated (0), positive (1) or missing (T).

Our resulting abstract domain is $Lab \rightarrow Cube_F$ where $Cube_F$ is the set of all possible cubes formed over the predicates in F .

Let's define $\sqsubseteq_{pa}, \sqcap_{pa}, \sqcup_{pa}$:

- Ordering \sqsubseteq_{pa} :
 - $a \sqsubseteq_{pa} b$ compares the elements pointwise
 - $a \sqsubseteq_{pa} b$ means $a \Rightarrow b$, but not vice versa
 - Examples:
 - $101 \sqsubseteq_{pa} 100 = no$
 - $100 \sqsubseteq_{pa} 101 = no$
 - $T01 \sqsubseteq_{pa} 10T = no$
 - $011 \sqsubseteq_{pa} 01T = yes$
 - $01T \sqsubseteq_{pa} 011 = no$
- Greatest lower bound \sqcap_{pa}
 - keep ϕ_k unselected ϕ_k appears in a , and $\neg\phi_k$ appears in b (or vice versa) in which case $a \sqcap_{pa} b$ returns false
 - Examples: $101 \sqcap_{pa} 100 = false$
 - $101 \sqcap_{pa} 10T = 101$
 - $T01 \sqcap_{pa} 10T = 101$
 - $00T \sqcap_{pa} 01T = false$
- Least upper bound \sqcup_{pa}
 - predicate (or negation) is kept if it appears in both cubes.
 - if all predicates are omitted then \sqcup_{pa} returns true
 - $a \vee b \Rightarrow a \sqcup_{pa} b$
 - Examples:
 - $101 \sqcup_{pa} 100 = 10T$
 - $101 \sqcup_{pa} 010 = true$
 - $T01 \sqcup_{pa} 10T = T0T$
 - $true \sqcup_{pa} 011 = true$

Connection Domains - Galois Connections

Next we setup a Galois connection between FOL and $Cube_F$ Concretization:

$\gamma : Cube_F \rightarrow FOL$

$\gamma(cube) = cube$

α is defined by γ and vice versa. So:

$\alpha(\psi) = \sqcap_{pa} \{cube | \psi \sqsubseteq_{FOL} \gamma(cube)\}$ **TODO:** where is this definition from?

In FOL, \sqsubseteq_{FOL} is: $\alpha(\psi) = \sqcap_{pa} \{cube | \psi \Rightarrow \gamma(cube)\}$ By substitution of γ we get:

$\alpha(\psi) = \sqcap_{pa} \{cube | \psi \Rightarrow cube\}$

Example:

$F = \{(2x - y), (y * y > 4), (x + y < 5)\}$

$\alpha(x = 1 \wedge y = 3) = \sqcap_{pa} \{0TT, T1T, TT1, 01T, 0T1, T11, 011\} = 011 = \neg(2x - y >$

$0) \wedge (y * y > 4) \wedge x + y < 5$

Optimization: instead of $\alpha(\psi) = \sqcap_{pa} \{cube | \psi \Rightarrow cube\}$ use $\alpha(\psi) = \sqcap_{pa} \{literal | \psi \Rightarrow literal\}$ $literal \iff cube$ has only one value different that T in the corresponding array. Example: 0TT, T1T. **TODO: no idea here, page 28**

TODO: why is this a galois connection?

8.3.2 Abstract Transformers

TODO: pretty cool construction of abstract transformer

$\llbracket action \rrbracket_{FOL}(\psi) = sp(\psi, action)$ is the best concrete transformer. **Why?** Now

lets defined the best abstract transformer: $\llbracket action \rrbracket_{pa} : Cube_F \rightarrow Cube_F$

$$\llbracket action \rrbracket_{pa}(c) = \alpha \circ \llbracket action \rrbracket_{FOL} \circ \gamma(c) \quad (8.1)$$

$$= \alpha \circ \llbracket action \rrbracket_{FOL}(c) \quad \text{substitution of } \gamma \quad (8.2)$$

$$= \sqcap_{pa} \{literal | \llbracket action \rrbracket_{FOL}(c) \Rightarrow literal\} \quad \text{definition of } \alpha \quad (8.3)$$

$$= \sqcap_{pa} \{literal | sp(c, action) \Rightarrow literal\} \quad \text{definition of } \llbracket action \rrbracket_{pa} \quad (8.4)$$

$$= \sqcap_{pa} \{literal | c \Rightarrow wp(literal, action)\} \quad \text{by sp to wp connection} \quad (8.5)$$

So to compute the transformer, we need to check whether the formula $c \Rightarrow wp(literal, action)$ is valid. This is done with an SMT solver and it's therefore desirable that the formulas are in some decidable logical fragment.

Key points

- for predicate abstraction the transformers are proved correct once and for all
- then we can instantiate predicate abstraction with any predicates and don't need to prove correctness
- in that sense predicate abstraction is a parametric framework
- the challenge is to find the sufficient predicates

8.3.3 Iterate to a fixed point

see page 33.

8.3.4 Summary

- we used wp for the best transformer because it introduces no quantifiers
- Best abstract transformer is sound by construction
- an important challenge is to find the sufficient predicates F to verify the program

An alternative to fixpoint iteration

TODO: important / good idea?

Input: a program P , a set of predicates F and a property S to verify.

1. Build a boolean program $B(P, F)$
Program contains only boolean variables, one for each predicate in F .
2. Check that $B(P, F)$ verifies the property S .
If the property holds for $B(P, F)$ then it also holds for P .

TODO: pages 36-37

8.4 Predicate Abstraction for Modern Concurrency

TODO: check again. especially the "boolean program" part. cube is $O(3^{length})$
used extrapolation to work on cubes $\ll cube$

Chapter 9

Synthesis from Examples

The goal of synthesis is to learn a program from examples. The key challenge is generalization, where we want to generalize from examples to something that is applicable in new situations. How do we generalize from a small number of examples? When do we know we're done?

9.1 Problem Dimensions

- Programming languages
Need a language expressive enough to capture programs of interest and is amenable to learning.
- Machine Learning
Need to learn a function in the language
- HCI
Input-output based interactive model

9.2 Version Spaces

First some keywords:

- hypothesis h : is a function $h : Input \rightarrow Output$
- hypothesis space H : is a set of hypotheses
- a hypothesis h is consistent with a sequence of input-output examples iff $\forall (x, y) \in D : h(x) = y$
- version space $VS_{H,D}$ consists of only those hypotheses in H that are consistent with all examples in D . $VS_{H,D} \subseteq H$

Version spaces are usually associated with some form of partial order on the hypotheses in $VS_{H,D}$. $h_1 \sqsubseteq h_2 \iff h_2$ covers more examples than h_1

Version spaces can be represented using just

- the most general consistent hypothesis (least upper bound)
- the most specific consistent hypothesis (greatest lower bound)

9.3 Version Space Algebra

Let's define operations on a version space $VS_{H,D}$

- combining version spaces
- joining version spaces
- transforming version spaces

This allows us to build complex version spaces out of simpler ones. Transformations are needed because given more examples we'll need to update the version space.

9.3.1 Union

$VS_{H1,D} \cup VS_{H2,D} = VS_{H1 \cup H2, D}$ For the same set of examples D . The functions in $H1$ and $H2$ have the same domains and ranges.

9.3.2 Join

Used symbol for join: \bowtie

$VS_{H1,D1} \bowtie VS_{H2,D2} = \{ \langle h_1, h_2 \rangle \mid h_1 \in VS_{H1,D1}, h_2 \in VS_{H2,D2}, C(\langle h_1, h_2 \rangle, D) \}$
Where $D1, D2$ are sequences of n input-output examples over $H1$ or $H2$ respectively.

D is a sequence $\langle (i_1, o_1), (k_1, l_1) \rangle \dots \langle (i_n, o_n), (k_n, l_n) \rangle$

$C(h, D)$ is a consistency predicate, true when hypothesis h consistent in D .

9.3.3 Independent join

Same definition but condition for independence:

$\forall h_1 \in H1, h_2 \in H2. C(h_1, D1) \wedge C(h_2, D2) \implies (\langle h_1, h_2 \rangle, D)$ That's essentially a cartesian product.

Efficiency of independent join \bowtie :

$Space(VS_{H1,D1} \bowtie VS_{H2,D2}) = Space(VS_{H1,D1}) + Space(VS_{H2,D2}) + C1$

$\forall d_1, d_2. Time(VS_{H1,D1} \bowtie VS_{H2,D2}, \langle d_1, d_2 \rangle) = Time(VS_{H1,D1}, d_1) + Time(VS_{H2,D2}, d_2) + C2$

9.3.4 Transforms

Transform one version space into another. Version space $VS1$ is a transform of version space $VS2$ iff: Let $VS1 = \{g \mid \exists f \in VS2. \forall i. g(i) = rm^{-1}(f(dm(i)))\}$. Where dm is a map from elements in the domain of functions in $VS1$ to elements in the domain of the functions in $VS2$ and rm is a 1-1 map from elements in the range of functions in $VS1$ to elements in the range of the functions in $VS2$.

9.4 Learning Version spaces

PAC learnability - probably approximately correct learning - helps us to answer the question "How many examples do we need to learn the target function?"

We run a version space on an input by applying every function in VS on that

input and collecting the outputs. Two hypotheses h_1, h_2 may "agree" on some input.

- Assign a probability to each hypothesis in the version space
- Choosing a probability allows us
 - to find better hypotheses (those with higher probability)
 - to specify priors
 - to deal with noisy data (by assignin a low probability < 0 to inconsistent hypotheses)
- $Pr(h|H)$ denotes the probability associated with hypothesis h in a hypothesis space H
- $Pr(V_i|W)$ denotes the prob. of a version space V_i given the version space W which is union of other version spaces
- if no prior knowledge, these are the uniform distributions
- The prob. of a hypothesis h in a version space V , denoted by $Pr(h|V)$ is updated as the version space changes and inconsistent hypotheses are removed.

9.4.1 Inductive Definitions of Probabilities

- initially: $Pr(h|V) = Pr(h|H)$
- **Transform:** if version space V_1 is a transform of another version space V_2 , then $Pr(h|V_1) = Pr(f, V_2)$ with f being the "matching function"
- **Union:** $P(h|V) = w_i * P(h|V_i)$
 V is the union of V_i 's
 w_i is the probability of version space V_i
TODO, page 20: werden die aufsummiert oder was?
- **Join:** $P(h|V)$ is the joint probability of the hypotheses h_1, \dots, h_n .
 if independent then: $P(h|V) = P(h_1|V_1) * \dots P(h_n|V_n)$

Having all those probabilities allows us to sum up all hypotheses that give the same result for a given input.

$$Pr_v^O(i) = \sum_{\forall h|h(i)=o} Pr(h|V) \quad (9.1)$$

This way we can present to the user the output states with the highest probability.

9.5 Example: SMARTedit

SMARTedit (by Tessa Lau) is one of the first working approaches and for learning interesting programs. It introduces version space algebras (VSA) for learning programs. SMARTedit is a texteditor that represents its editor state as $\sigma = \langle T, L, P, E \rangle$ with T being the contents of the text buffer, L the cursor location (row, column), P the contents of the clipboard and E a contiguous region of T representing selected text.

9.5.1 Action

An editor action is a function $a : \Sigma \rightarrow \Sigma$ with Σ being the set of possible editor states.

Example:

$\langle T, (42, 0), P, E \rangle \rightarrow \langle T, (43, 0), P, E \rangle$ "move to the next line" or "move to the beginning of line 43" are consistent.

"move to the beginning of line 47" and "move to the end of line 41" are inconsistent. Actions have locations. E.g move to location, delete between two locations etc.

9.5.2 Version space

SMARTedit's version space is a union of different kinds of actions. The action function maps from one state to another.

Location version space

TODO page 36

9.5.3 Learning

The system learns by updating the version space on new examples. Inconsistent hypotheses are removed and parts of the hierarchy are pruned away. An examples consistency is tested against the entire version space. **see example on page 40**

9.5.4 string searching version spaces

Whatever... **TODO**

Chapter 10

Dynamic race detection

10.1 Race conditions

Data races: potential atomicity bug

General races: potential determinism bug