Abstract Interpretation

Jan Midtgaard

Week 1

http://www.cs.au.dk/~jmi/AbsInt/

Aarhus University, Q4 - 2012

What is this course about?

Crudely simplified the history of program analysis (or static analysis) can be split in two:

- □ an American school of program analysis
- □ a French school of program analysis

I highly recommend the *Static Analysis course*, which gives a nice introduction mainly to the American approach.

This course is concerned with the alternative, French approach.

Which is the right approach?

None of them is right or wrong — it is simply an alternative view — an eye opener to a new world.

It can be used to explain existing approaches and extend or strengthen them

In 7 weeks, you will be in a position to make an informed opinion

It is not just an academic theory: it has been used to check/verify flight control software for both Airbus and Mars missions. By the end of this course, we will read papers about those.

It will get bloody — there will be mathematics — there will be semantics

You take the red pill. . .

You take the red pill...

...you stay in Wonderland and I show you how deep the rabbit-hole goes...

What is abstract interpretation?

- It is a theory of *semantics-based program analysis*
- It was initially conceived in the late 1970's by Patrick and Radhia Cousot
- It has been refined over the last 40 years
  - to new applications
  - to new kinds of semantics
  - to new programming paradigms
  - by new abstract domains
  - ...

Learning outcomes and competences

The participants must at the end of the course be able to:

- *describe* and *explain* basic analyses in terms of classical abstract interpretation.
- *apply* and *reason* about Galois connections.
- *implement* abstract interpreters on the basis of the derived program analyses.

Pedagogical choices / Contract

Lectures - sometimes including a few exercises in class

Reading - read research papers and slides

Assignments - both mathematics and programming.
They are mandatory

Project - a chance for you to apply your newly acquired skills to a topic of your choice (both mathematics and programming, preferably)

Exam - explain to us how you applied your newly acquired skills, and
we'll have an informed discussion of the outcome

Your background

I’m assuming you all have followed Programming Languages (dProgSprog) and Compilation (dOvs).

How many of you have followed

- □ the *Static Analysis* course?
- □ Olivier’s *IFP* / *TFP* courses?
- □ a *semantics* course?
  - by Claus Brabrand or Jakob Andersen?
  - by Klaus Ostermann?
- □ Glynn Winskel’s *set theory* course?

Outline

- What and how of the course
- Transition systems
- Math: Posets, CPOs, complete lattices, Galois connections, fixed points
- Abstract interpretation basics
- OCaml intro

Transition systems

Transition systems - quick recap

You already know transition systems from dADS 1.

Definition. A transition system is a triple (quadruple) $\langle S, I, F, \rightarrow \rangle$ where

- □ S is a set of states
- □ $I \subseteq S$ is a set of initial states
- □ $F \subseteq S$ is an optional set of final states ($\forall s \in F, s' \in S : s \not\rightarrow s'$)
- □ $\rightarrow \subseteq S \times S$ is a transition relation relating a state to its (possible) successors

Example 1: Euclid's algorithm

Given two numbers $x, y \in \mathbb{N}$ we can describe Euclid's GCD algorithm as a transition system:

$$\begin{aligned}
S &= \mathbb{N} \times \mathbb{N} \\
I &= \{\langle x,\, y\rangle\} \\
F &= \{\langle n,\, n\rangle \mid n \in \mathbb{N}\} \\
\rightarrow : \langle n,\, m\rangle &\rightarrow \langle n-m,\, m\rangle && \text{if } \ n > m \\
\langle n,\, m\rangle &\rightarrow \langle n,\, m-n\rangle && \text{if } \ n < m
\end{aligned}$$

where we have written the transition relation using *infix notation*.
We can write it even more formally as:

$$\begin{aligned}
\rightarrow = &\{(\langle n,\, m\rangle, \langle n-m,\, m\rangle) \mid n > m\} \\
&\cup \{(\langle n,\, m\rangle, \langle n,\, m-n\rangle) \mid n < m\}
\end{aligned}$$

Example 2: Modeling a program

Modeling the program

```
x := 0;
while (x < 100) {
    x := x + 1;
}
```

as a transition system:

$$
\begin{aligned}
S &= \mathbb{Z} \\
I &= \{0\} \\
\rightarrow &= \{(x, x') \mid x < 100 \ \wedge \ x' = x + 1\}
\end{aligned}
$$

How to get from a program to a transition system is the topic of next week's lecture.

For now we assume that we can model the semantics (the meaning) of a program as a transition system.

Mathematical foundations

Partially ordered sets

Definition. A partially ordered set (poset) $\langle S; \sqsubseteq \rangle$ is a set S equipped with a binary relation $\sqsubseteq \subseteq S \times S$ with the following properties:

- Reflexive: $\forall a \in S : a \sqsubseteq a$
- Antisymmetric: $\forall a, b \in S : a \sqsubseteq b \ \wedge \ b \sqsubseteq a \implies a = b$
- Transitive: $\forall a, b, c \in S : a \sqsubseteq b \ \wedge \ b \sqsubseteq c \implies a \sqsubseteq c$

Example 1: $\langle \mathbb{N}; \leq \rangle$ is a poset

Example 2: $\langle \wp(S); \subseteq \rangle$ is a poset

Note: $\wp(S)$ is sometimes written 2^S

Upper and lower bounds

Let $\langle P; \sqsubseteq \rangle$ be a partially ordered set.

Definition. $u \in P$ is an *upper bound* of $S \subseteq P$ iff $\forall s \in S : s \sqsubseteq u$

Definition. $l \in P$ is an *lower bound* of $S \subseteq P$ iff $\forall s \in S : l \sqsubseteq s$

Definition. $u \in P$ is a *least upper bound* (lub) of $S \subseteq P$ iff it is an upper bound of S and it is less than all other upper bounds: $\forall u' \in P : (\forall s \in S : s \sqsubseteq u') \implies u \sqsubseteq u'$

Definition. $l \in P$ is a *greatest lower bound* (glb) of $S \subseteq P$ iff it is an lower bound of S and it is greater than all other lower bounds: $\forall l' \in P : (\forall s \in S : l' \sqsubseteq s) \implies l' \sqsubseteq l$

Complete Partial Orders (CPOs)

Definition. A complete partial order is a poset such that all increasing chains $c_i, i \in \mathbb{N}$ ($\forall i \in \mathbb{N} : c_i \sqsubseteq c_{i+1}$) have a least upper bound:

$$\bigsqcup_{i \in \mathbb{N}} c_i$$

Non-example: $\langle \mathbb{N}; \leq \rangle$ is *not* a CPO. Why?

Example: $\langle \wp(S); \subseteq \rangle$ is a CPO.

Complete lattices

Definition. A complete lattice is a poset $\langle C; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ such that

- the least upper bound $\sqcup S$ and
- the greatest lower bound $\sqcap S$ exists for every subset S of C.
- $\bot = \sqcap C$ denotes the infimum of C and
- $\top = \sqcup C$ denotes the supremum of C.

Example 1: $\langle \wp(S); \subseteq, \emptyset, S, \cup, \cap \rangle$ is a complete lattice.

Example 2: The integers (extended with $-\infty$ and $+\infty$) is a complete lattice $\langle \mathbb{Z} \cup \{-\infty, +\infty\}; \leq, -\infty, +\infty, \max, \min \rangle$.

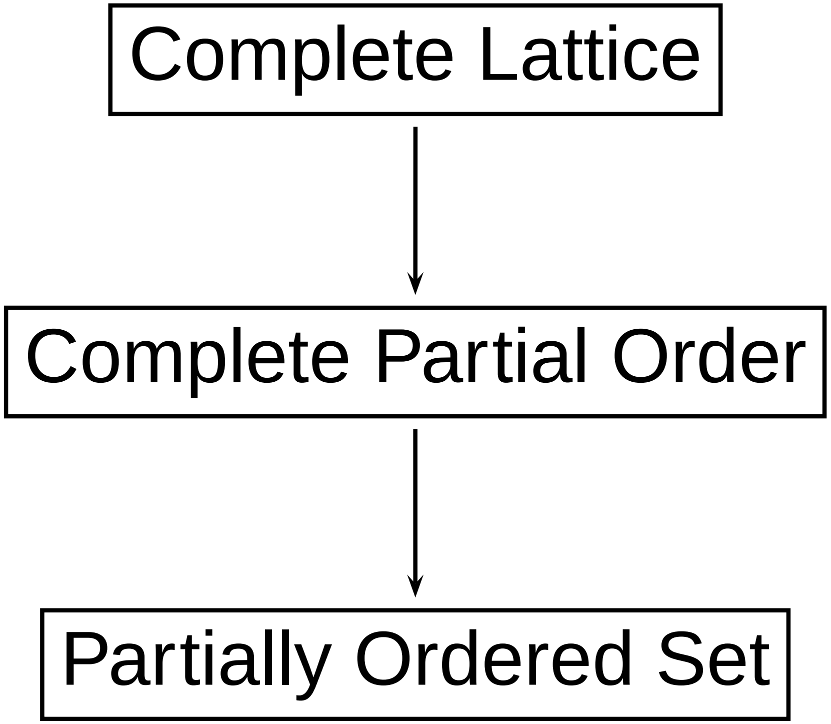
Example: A complete lattice of functions

Theorem. The set of total functions $D \to C$, whose codomain is a complete lattice $\langle C; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$, is itself a complete lattice $\langle D \to C; \dot{\sqsubseteq}, \dot{\bot}, \dot{\top}, \dot{\sqcup}, \dot{\sqcap} \rangle$ under the pointwise ordering $f \mathrel{\dot{\sqsubseteq}} f' \iff \forall x. f(x) \sqsubseteq f'(x)$, and with

- $\dot{\bot} = \lambda x. \bot$
- $\dot{\top} = \lambda x. \top$
- $f \mathbin{\dot{\sqcup}} g = \lambda x. f(x) \sqcup g(x)$
- $f \mathbin{\dot{\sqcap}} g = \lambda x. f(x) \sqcap g(x)$

Here $\lambda x. \ldots$ is a mathematical function with argument x.

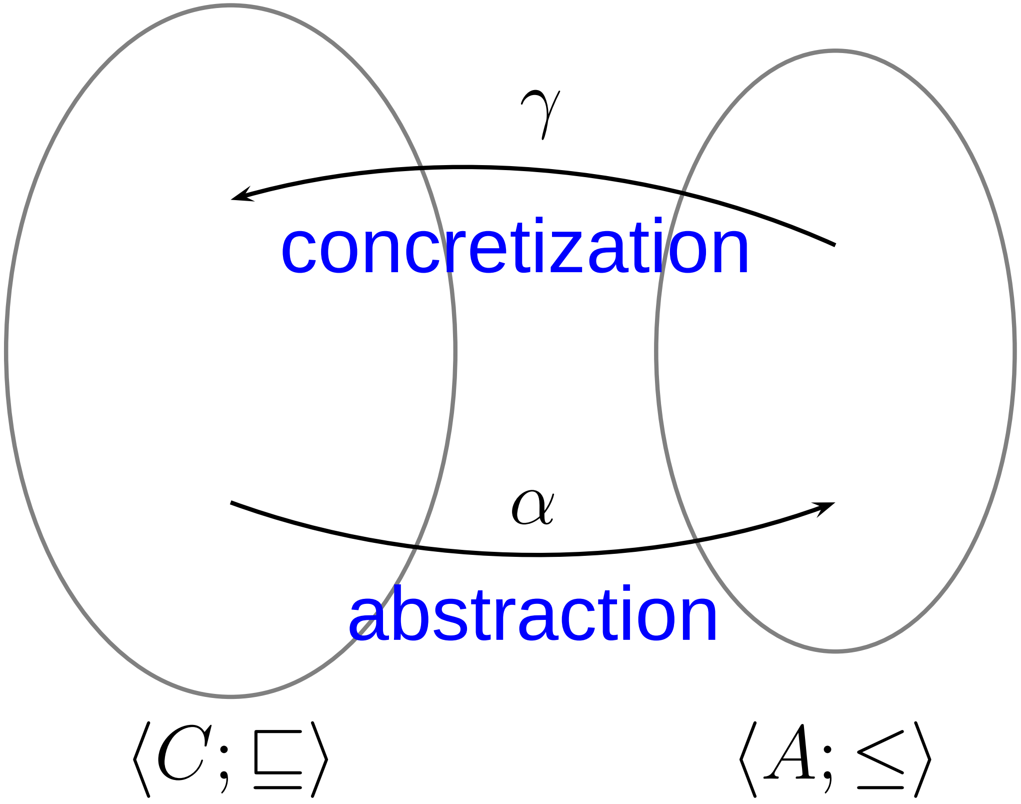
A quick comparison



Galois connections

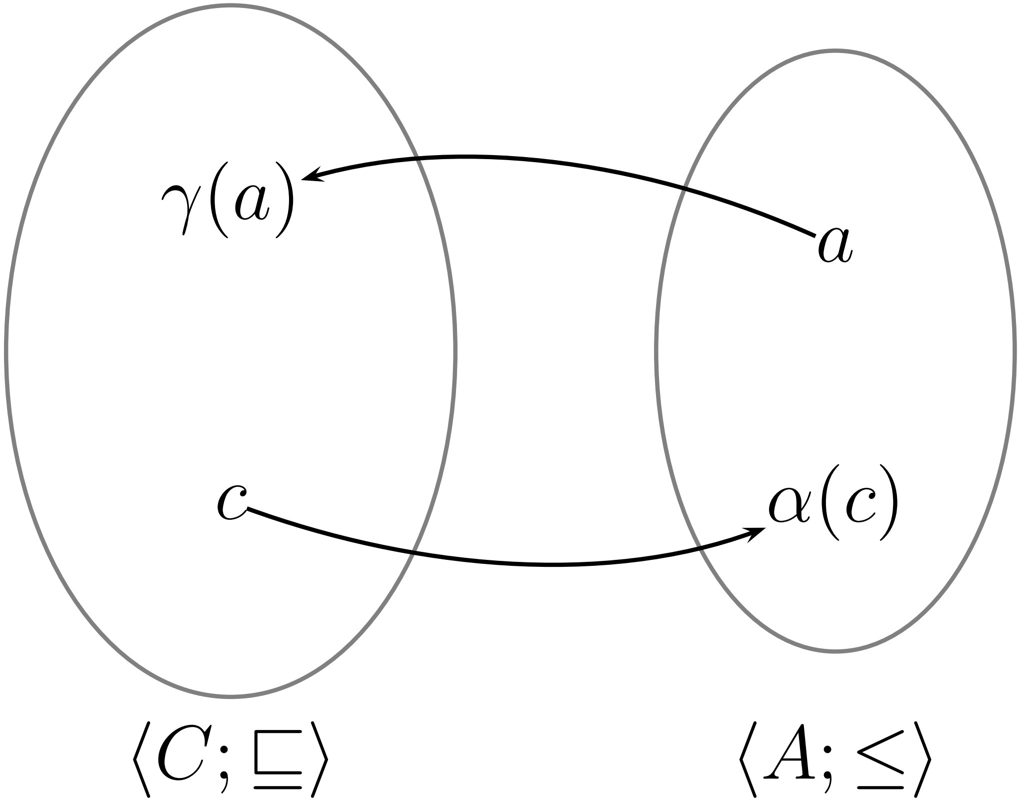
Galois connections

Definition. A Galois connection is a pair of functions α, γ between two partially ordered sets:



Galois connections

Definition. A Galois connection is a pair of functions α, γ between two partially ordered sets:



such that: $\forall a \in A, c \in C : \alpha(c) \leq a \iff c \sqsubseteq \gamma(a)$

A familiar example

You already know the pattern of moving from one side of an inequation to another from high school:

$$\forall x, y, z \in \mathbb{Z} \ : \ x + z \leq y \iff x \leq y - z$$

which we can write with α and γ as:

$$\begin{aligned} \forall x, y, z \in \mathbb{Z} \ : \ \alpha(x) \leq y \iff x \leq \gamma(y) \\ \text{where } \alpha(n) = n + z \\ \gamma(n) = n - z \end{aligned}$$

An equivalent definition

Definition. A Galois connection is a pair of functions α and γ satisfying

(a) α and γ are monotone
(for all $c, c' \in C : c \sqsubseteq c' \implies \alpha(c) \leq \alpha(c')$ and
for all $a, a' \in A : a \leq a' \implies \gamma(a) \sqsubseteq \gamma(a')$),

(b) $\alpha \circ \gamma$ is reductive (for all $a \in A : \alpha \circ \gamma(a) \leq a$),

(c) $\gamma \circ \alpha$ is extensive (for all $c \in C : c \sqsubseteq \gamma \circ \alpha(c)$).

Galois connections are typeset as $\langle C; \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \leq \rangle$.

Galois connection properties (1/3)

Theorem. For a Galois connection between two complete lattices $\langle C; \sqsubseteq, \bot_c, \top_c, \sqcup, \sqcap \rangle$ and $\langle A; \leq, \bot_a, \top_a, \vee, \wedge \rangle$, α is a complete join-morphism (CJM):

for all $S_c \subseteq C : \alpha(\sqcup S_c) = \vee \alpha(S_c) = \vee \{\alpha(c) \mid c \in S_c\}$

and γ is a complete meet morphism (CMM):

for all $S_a \subseteq A : \gamma(\wedge S_a) = \sqcap \gamma(S_a) = \sqcap \{\gamma(a) \mid a \in S_a\}$

Galois connection properties (2/3)

Theorem. The composition of two Galois connections $\langle C; \sqsubseteq\rangle \xleftarrow[\alpha_1]{\gamma_1} \langle B; \subseteq\rangle$ and $\langle B; \subseteq\rangle \xleftarrow[\alpha_2]{\gamma_2} \langle A; \leq\rangle$ is itself a Galois connection:

$$\langle C; \sqsubseteq\rangle \xleftarrow[\alpha_2 \circ \alpha_1]{\gamma_1 \circ \gamma_2} \langle A; \leq\rangle$$

We can typeset this theorem as an inference rule:

$$\frac{\langle C; \sqsubseteq\rangle \xleftarrow[\alpha_1]{\gamma_1} \langle B; \subseteq\rangle \qquad \langle B; \subseteq\rangle \xleftarrow[\alpha_2]{\gamma_2} \langle A; \leq\rangle}{\langle C; \sqsubseteq\rangle \xleftarrow[\alpha_2 \circ \alpha_1]{\gamma_1 \circ \gamma_2} \langle A; \leq\rangle}$$

Hence Galois connections stack up like Lego bricks!

Galois connection properties (3/3)

Galois connections in which α is surjective / onto (or equivalently γ is injective) are typeset as:

$$\langle C; \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftarrows\!\!\!\!\twoheadrightarrow}} \langle A; \leq \rangle$$

and sometimes called Galois surjections (or insertions)

Galois connections in which α is injective / one-to-one (or equivalently γ is surjective) are typeset as:

$$\langle C; \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow\!\!\!\!\rightarrows}} \langle A; \leq \rangle$$

and sometimes called Galois injections

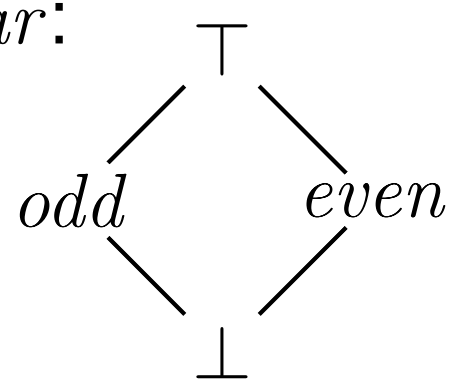
When both α and γ are surjective, the two domains are isomorphic.

Example: The Parity abstract domain

Consider the abstraction into the Parity domain:

$$\langle \wp(\mathbb{N}_0); \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle Par; \sqsubseteq \rangle$$

Par:



The above *Hasse diagram* defines the Parity ordering.

The abstraction and concretization functions are:

$$\gamma(P) = \begin{cases} \emptyset & \text{if } P = \bot \\ \{n \in \mathbb{N}_0 \mid n \text{ is odd}\} & \text{if } P = odd \\ \{n \in \mathbb{N}_0 \mid n \text{ is even}\} & \text{if } P = even \\ \mathbb{N}_0 & \text{if } P = \top \end{cases} \qquad \alpha(N) = \begin{cases} \bot & \text{if } N = \emptyset \\ odd & \text{if } \forall n \in N : n \text{ is odd} \\ even & \text{if } \forall n \in N : n \text{ is even} \\ \top & \text{otherwise} \end{cases}$$

Example: an isomorphism

We can represent a set of pairs as a function from a first component to second components:

$$\langle \wp(A \times B); \subseteq \rangle \xleftrightarrow[\alpha]{\gamma} \langle A \to \wp(B); \dot{\subseteq} \rangle$$

where $\begin{aligned} \alpha(R) &= \lambda a.\{b \mid (a, b) \in R\} \\ \gamma(F) &= \{(a, b) \mid b \in F(a)\} \end{aligned}$

Fixed points

Fixed points, briefly

Definition. a *fixed point* of a function f, is a point x such that $f(x) = x$

Assume $f : P \to P$ operates over a poset $\langle P; \sqsubseteq \rangle$

Definition. a *pre-fixed point* is a point x such that $x \sqsubseteq f(x)$

Definition. a *post-fixed point* is a point x such that $f(x) \sqsubseteq x$

Definition. a *least fixed point* (lfp) is a fixed point l such that for all other fixed points $l' : (f(l') = l') \implies l \sqsubseteq l'$

Definition. a *greatest fixed point* (gfp) is a fixed point l such that for all other fixed points $l' : (f(l') = l') \implies l' \sqsubseteq l$

Tarski's fixed point theorem

Theorem. If L is a complete lattice and $f : L \to L$ is a monotone function, f's fixed points themselves form a complete lattice.

Hence Tarski tells us that *there exists a least fixed point*.

Abstract interpretation basics

Abstract interpretation basics

Canonical abstract interpretation approximates the *collecting semantics* of a transition system.

A standard example of a collecting semantics is the *reachable states* from a given set of initial states I.
Given a transition function T defined as:

$$T(\Sigma) = I \cup \{\sigma \mid \exists \sigma' \in \Sigma : \sigma' \to \sigma\}$$

we can express the reachable states of T as the least fixed point $\operatorname{lfp} T$ of T.
For a fixed point $T(\Sigma) = \Sigma$ of T:

$$I \subseteq \Sigma \quad \wedge \quad \forall \sigma' \in \Sigma : \sigma' \to \sigma \implies \sigma \in \Sigma$$

which expresses the transitive closure of the states reachable from I.

Abstract interpretation basics

Canonical abstract interpretation approximates the *collecting semantics* of a transition system.

A standard example of a collecting semantics is the *reachable states* from a given set of initial states I.
Given a transition function T defined as:

$$T(\Sigma) = I \cup \{\sigma \mid \exists \sigma' \in \Sigma : \sigma' \to \sigma\}$$

we can express the reachable states of T as the least fixed point $\operatorname{lfp} T$ of T.
We can compute $\operatorname{lfp} T$ by Kleene iteration[1]:

$$\bot, T(\bot), T^2(\bot), T^3(\bot), \ldots$$

[1] In general we can only compute $\operatorname{lfp} f$ if f is contiguous $f(\sqcup S) = \sqcup f(S)$

The strength of the collecting semantics

- The collecting semantics is ideal, i.e., it is the *most precise analysis*.
- Unfortunately it is in general uncomputable: it is as hard as interpreting (i.e., running) a program
- We therefore approximate the collecting semantics, by computing a fixed point over an alternative and perhaps simpler domain: an *abstract* interpretation

Abstraction and analysis using Galois connections

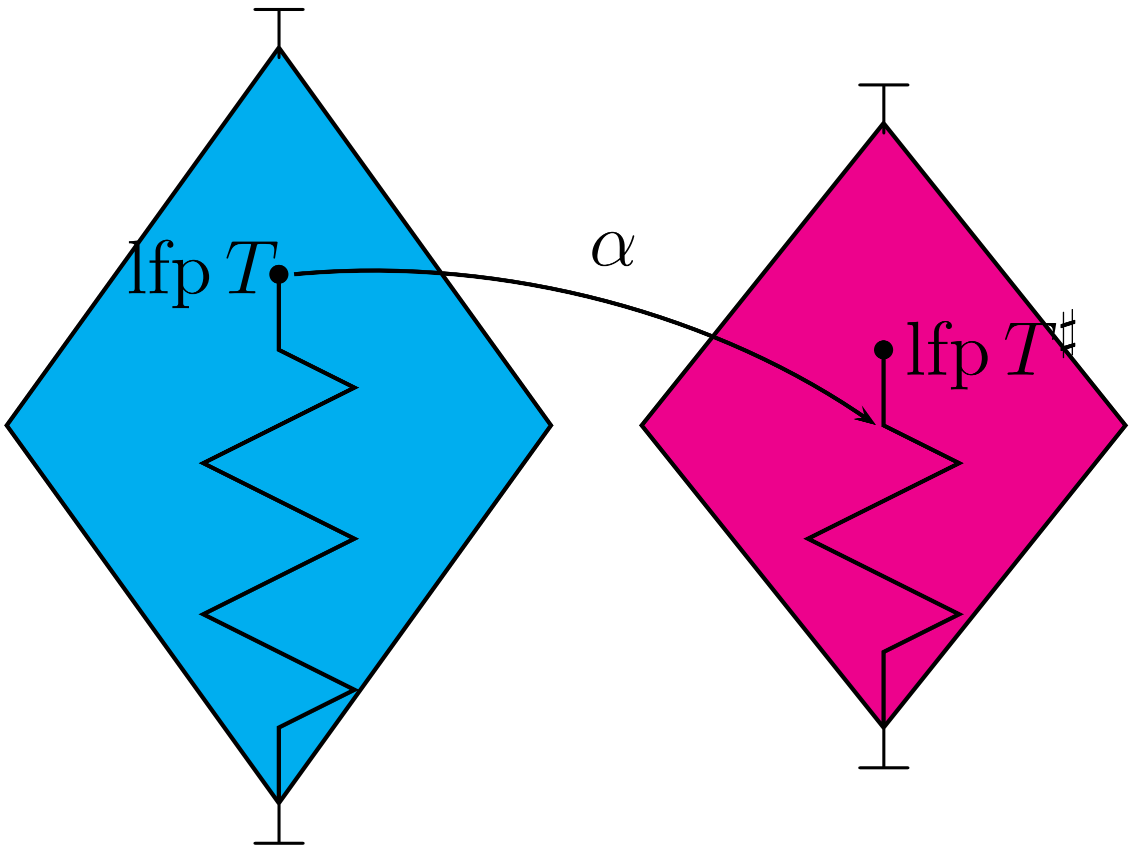
Abstractions are represented as Galois connections which connect complete lattices through α and γ.

We can derive an analysis systematically by composing the transition function with these functions: $\alpha \circ T \circ \gamma$ and gradually refine the collecting semantics into a computable analysis function by mere calculation.

Hence instead of *inventing* a static analysis, we arrive at one by a *structured abstraction* of the set of states $\wp(S)$.

Galois connection-based analysis

By the *fixed point transfer theorem* we can compute a sound approximation of the collecting semantics:



Theorem. Let $\langle C; \sqsubseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A; \leq \rangle$ be a Galois connection between complete lattices. If T and $T^\sharp$ are monotone and $\alpha \circ T \circ \gamma \mathrel{\dot{\leq}} T^\sharp$ then $\alpha(\mathrm{lfp}\, T) \leq \mathrm{lfp}\, T^\sharp$

Variations

An alternative approach

Rather than simplifying the abstract domains into finite ones, *widening* and *narrowing* permits infinite ones.

A first widening iteration overshoots the least fixed point but still ensures termination.

A second narrowing iteration improves the results of the widening iteration.

Widening

We compute instead the limit of the sequence:

$$X_0 = \bot$$
$$X_{i+1} = X_i \triangledown T(X_i)$$

where $\triangledown$ denotes the *widening operator*: an operator with the following properties:

- For all $x, y : x \sqsubseteq (x \triangledown y) \ \wedge \ y \sqsubseteq (x \triangledown y)$
- For any increasing chain $Y_0 \sqsubseteq Y_1 \sqsubseteq Y_2 \sqsubseteq \ldots$ the alternative chain defined as $Y'_0 = Y_0$ and $Y'_{i+1} = Y'_i \triangledown Y_{i+1}$ stabilizes after a finite amount of steps.

Narrowing

We can compute the limit of the sequence:

$$X_0 = \lim_i Y_i$$
$$X_{i+1} = X_i \vartriangle T(X_i)$$

where $\vartriangle$ denotes the *narrowing operator*: an operator with the following properties:

- For all $x, y : (x \vartriangle y) \sqsubseteq x$
- For all $x, y, z : (x \sqsubseteq y \ \wedge \ x \sqsubseteq z) \implies x \sqsubseteq (y \vartriangle z)$
- For any chain Y_i the alternative chain defined as $Y'_0 = Y_0$ and $Y'_{i+1} = Y'_i \vartriangle Y_{i+1}$ stabilizes after a finite amount of steps.

Some words on OCaml

Why OCaml?

In this course we will use the OCaml programming language

Why OCaml?

In this course we will use the OCaml programming language

Why?

Why OCaml?

In this course we will use the OCaml programming language

Why?

→ It's a good opportunity to learn a new language and add it to your CV

- □ The core of Microsoft's F# is based on OCaml

→ It's a good fit for the job

- □ Microsoft's static device driver verifier is written in OCaml
- □ ASTREÉ is written in OCaml

OCaml is an ML dialect

Hence it

- is expression-based, hence everything has a value
- is strongly typed
- is statically scoped
- has algebraic datatypes, lists, tuples, and pattern matching
- has higher-order functions
- ...

In addition it includes some object-oriented extensions (hence the O in OCaml).

Compilers and IDEs

There is both

- a bytecode compiler (`ocamlc`) and
- an optimizing native code compiler (`ocamlopt`)

freely available for many platforms.

- For emacs I recommend tuareg-mode
- For Eclipse people recommend: OCaIDE

  `http://www.algo-prog.info/ocaide/`
  `http://www.cs.jhu.edu/~scott/pl/caml/ocaide.shtml`

- For VIM: OMLet
- For _: please let me know of your findings

SML/Scheme vs OCaml (1/2)

You all know SML or Scheme from ProgSprog, so we will focus on the differences.

Instead of `fun foo x = ...`

or `(define (foo x) ...)`

we write `let foo x = ...`

Catch 0: function application binds stronger than addition: Hence `f x+1` means `(f x)+1`

Catch 1: recursive functions must be marked '`rec`':

```
let rec fac n = match n with
  | 0 -> 1
  | n -> n * fac (n - 1)
```

SML/Scheme vs OCaml (2/2)

Like in SML and Scheme `let` is also used for local declarations ([ ] is nil, : : is cons):

```
let concat xs ys =
  let rec walk xs = match xs with
    | [] -> ys
    | x::xs' -> x::(walk xs')
  in
  walk xs
```

however without an `end` to end the block.

Note also that OCaml uses `match ... with` instead of SML's `case ... of`.

Exercise: write in OCaml a function `sumlist` of type

```
sumlist : int list -> int
```

Catches and Gotchas

Catch 2: Semicolon '`;`' separates list elements (rather than comma '`,`'). For example, compare the types of `[1,2,3]` and `[1;2;3]`

Tuples (and pairs) can be written without parens!

Catch 3: datatype constructors must be capitalized

```
type 'a tree = Leaf of 'a
             | Node of 'a tree * 'a tree
```

anything else is a parse error!

Catch 4: The evaluation order is unspecified — however the compiler uses right-to-left in practice(!)

OCaml modules

OCaml has a powerful module system with

- signatures (think interface) and
- functors (think `module -> module` function)

Example:

```
module Intset =
  Set.Make (struct
              type t = ...   (* element type *)
              let compare = ...
                        (* element comparison *)

            end)
```

OCaml modules

OCaml has a powerful module system with

- signatures (think interface) and
- functors (think `module -> module` function)

Example:

```
module Intset =
  Set.Make (struct
              type t = int
              let compare n1 n2 =
                if n1 == n2 then 0 else
                  if n1 > n2 then 1 else -1
            end)
```

OCaml modules

OCaml has a powerful module system with

- signatures (think interface) and
- functors (think `module -> module` function)

Example:

```
module Intset =
  Set.Make (struct
              type t = int
              let compare n1 n2 =
                if n1 == n2 then 0 else
                  if n1 > n2 then 1 else -1
            end)
```

Builtin maps are similar:

```
module Mymap = Map.Make(struct ... end)
```

OCaml modules and separate compilation

We can separate the implementation and the interface of a module into two separate files `x.ml` and `x.mli`.

This is equivalent to

```
module X: sig (* contents of file x.mli *) end
        = struct (* contents of file x.ml *) end
```

Catch 5: Files are lower-case, but their modules are capitalized. Hence, the module in file `set.ml` is referred to as `Set`.

If we write

```
module S = struct let f = ... end
```

in a file `foo.ml` then we (need to) refer to `f` as `Foo.S.f`

Relevant links

- SML/OCaml comparisons by Rossberg and Chlipala

  `http://www.mpi-sws.org/~rossberg/sml-vs-ocaml.html`

  `http://adam.chlipala.net/mlcomp/`

- OCaml reference manual

  `http://caml.inria.fr/pub/docs/manual-ocaml/`

- Standard library documentation

  `http://caml.inria.fr/pub/docs/manual-ocaml/libref/`

- Jason Hickey's online book

  `http://files.metaprl.org/doc/ocaml-book.pdf`

- Two mailing lists (beginner + main list)

- . . .

Let's code something!

Let's implement

- a transition system interface,
- an instantiation thereof, and
- the transition function from the reachable states collecting semantics

Summary

Summary

We have covered

- The what and the how of the course
  - Remember the measure of success: *an application of AI*
  - So start thinking of a transition system for your project (Turing machine, Traveling Salesman,. . .)
- The basics of abstract interpretation (transition systems, reachable states collecting semantics, Galois connections, . . .)
- A crash course in OCaml