Program analysis — intro

http://perso.ens-lyon.fr/daniel.hirschkoff/cap/

Program analysis

- in CAP, we discuss programs manipulating programs

  compute something with a program as an input

  - another program
  - a property of the program *what it does (not)*

    $\hookrightarrow$ accept/reject, transform the initial program

- we shall focus on smaller scale languages

  1. small imperative language: IMP
     - 1.1 Abstract Interpretation (automatic, the program is the only input)
     - 1.2 Hoare triples (interaction with the user)
  2. small functional language: FUN
     - 2.1 type inference
     - 2.2 abstract machines and compilation
     - 2.3 intermediate representations

- in breadth rather than in depth
  - few proofs (see references on the www page)
- prerequisites: order theory, semantics

Abstract Interpretation

With material from the courses by A. Miné and P. Roux

1. The method

Analysing programs

- typical questions we want to aks / bugs we want to avoid

`x = a/b`	make sure `b`$\neq 0$
`x = t[i]`	make sure `i` is within the bounds of `t`
`i = i+1`	make sure there is no overflow

- Abstract Interpretation can also be used to perform more refined analyses

Runs of a program

an example of a program and its runs

`demo-concrete.pdf`

- we want to know what values a variable can have at a given point of the program
- we would like to *compute* this *(without any input from the user)*

on the board

Know everything about all possible runs of the program

- annotate nodes of (some kind of) Control Flow Graphs with labels $\ell \in \mathcal{L}$
- during execution, a program state (ℓ, σ) consists of
  - a control state $\ell \in \mathcal{L}$ and
  - an environment (memory state) $\sigma \in \mathcal{V} \to \mathbb{Z}$
- **concrete semantics** (*meaning*) of the program
  - write a recursive equation involving sets of environments
  - we are interested in **the least fixpoint** of some operator acting on $\mathcal{P}(\mathcal{V} \to \mathbb{Z})$
  - this fixpoint yields a function of type $\mathcal{L} \to \mathcal{P}(\mathcal{V} \to \mathbb{Z})$

    associating a set of possible stores (memory states) to every label in the program

the least fixpoit exists (Knaster-Tarski's theorem)

. . . **but there is no hope of computing it**

(either impossible/undecidable or too costly)

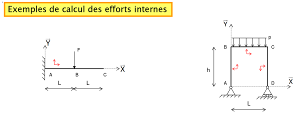
Computing an abstraction

"I took a speed reading course and read War and Peace in twenty minutes.

It involves Russia."

Let's get rough

- instead of computing the concrete semantics,
  compute an abstract semantics



- be less precise, and more computable

  scale down our ambitions, and strike a balance

- rough but sound

the abstract semantics
contains the concrete semantics

some examples of abstractions

demo-signs.pdf   demo-cstes.pdf

2. How it works

(and why — a glance at the mathematical justification)

on the board

The general method: $\mathcal{D}$ and $\mathcal{D}^\sharp$, via γ

the concretisation function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$

- γ should be monotone
- $a \in \mathcal{D}^\sharp$ is a *sound abstraction* of $c \in \mathcal{D}$ if $c \subseteq \gamma(a)$
- $g : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is a *sound abstraction* of $f : \mathcal{D} \to \mathcal{D}$
  if $\forall a \in \mathcal{D}^\sharp,\ (f \circ \gamma)(a) \subseteq (\gamma \circ g)(a)$

move from the concrete semantics to the abstract semantics:

from $R_\ell = \bigcup_{(j,c,\ell)\in A} [\![c]\!]\, R_j$ to $\sigma^\sharp_\ell = \bigcup^\sharp_{(j,c,\ell)\in A} [\![c]\!]^\sharp\, \sigma^\sharp_j$

- $\sigma^\sharp_\ell$, $\sigma^\sharp_j$: *abstract environments*
- $[\![\,\cdot\,]\!]^\sharp$: abstract transfer function

The answers of Abstract Interpretation

Theorem (Soundness): $\forall \ell \in \mathcal{L},\ R_\ell \subseteq \gamma(\sigma_\ell^\sharp)$.

because we use *sound* operators ($\cup^\sharp$, $+^\sharp$, ...) in $\mathcal{D}^\sharp$, we keep over-approximating when computing the abstract semantics

cf. talking with toddlers  

Abstract Interpretation: *compute* the abstract semantics, and check the required condition

- if the answer is “ok”, then it is “ok”

  for example, 0 *is not among the possible values for* X *at that point in the program*

- if the answer is “no”, then work needs to be done

Insuring that an answer is provided

we want *effective* computations

- everything should be computable in $\mathcal{D}^\sharp$
  - representation of elements of $\mathcal{D}^\sharp$
  - $\sqsubseteq^\sharp$, $\cup^\sharp$, ...
- computing the abstract semantics
  - computing $\sigma_\ell^\sharp$
    relies on the definition of abstract operators $+^\sharp, -^\sharp, \ldots$
  - **computing the least fixpoint**
    - Kleene iterations $\perp$, $F(\perp)$, $F(F(\perp)), \ldots$
    - a finite number of them: stabilisation
      . ok if the lattice is of finite height
      . otherwise...

Widening

- the analysis must be able to answer in *reasonable time*
- in some cases, the abstract domain $\mathcal{D}^\sharp$ is of unbounded height

  to guarantee convergence of the computation of the least fixpoint, we use a widening operator $\nabla : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ satisfying:
  - $\forall x^\sharp, y^\sharp, \quad x^\sharp \cup^\sharp y^\sharp \;\sqsubseteq^\sharp\; x^\sharp \nabla y^\sharp.$ soundness
  - for any sequence $(y_i^\sharp)_{i \geq 0}$,
    the sequence $x_0^\sharp = y_0^\sharp, \;\; x_{i+1}^\sharp = x_i^\sharp \nabla y_{i+1}^\sharp$ satisfies
    $\exists n.\, x_{n+1}^\sharp = x_n^\sharp.$ stablilisation

  ∇ "extrapolates"
- $\sigma_\ell^{\sharp\, n+1} \;=\; \sigma_\ell^{\sharp\, n} \;\nabla\; \bigcup^\sharp_{(j,c,\ell) \in A} [\![c]\!]^\sharp \;\sigma_j^{\sharp\, n}$

  for some nodes ℓ belonging to cycles in the CFG

  $\hookrightarrow$ convergence of the iteration
- a *narrowing operator* can also be used to make the analysis more precise after applying widening

TP next week

- you will be given a program that computes the abstract semantics according to a given value abstract domain
- you will define several value abstract domains, and see how the analysis of programs is affected

  you might want to write down equations
  before coding $\sqsubseteq^\sharp \quad +^\sharp \quad -^\sharp \ldots$

- all this in OCaml

  you don't need to be an expert OCaml programmer

  basically, define (simple) types, and (simple) functions acting on such types

- install OCaml on your laptop

Further notions in Abstract Interpretation

References

- course by Pierre Roux
  AI in 3 lessons
- course by Antoine Miné (and others)
  much more detailed and in depth (M2)

see links from the course webpage

- *many thanks to Antoine and Pierre for allowing me to use their material*
- a peculiarity in terminology:

  . prefixpoint $f(x) \sqsubseteq x$
  . postfixpoint $x \sqsubseteq f(x)$

  as seen, e.g., in L3IF

  ... they use the converse

Galois connections

α: monotone **abstraction function**

$$(\mathcal{D}, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq^\sharp)$$

$$\alpha(x) \sqsubseteq^\sharp y^\sharp \iff x \sqsubseteq \gamma(y^\sharp)$$

- any $x \in \mathcal{D}$ has a best abstraction $\alpha(x)$

Relational abstract domains

- the **concrete semantics** is given by a function (which is difficult to compute) in $\mathcal{L} \to \mathcal{P}(\mathcal{V} \to \mathbb{Z})$

  associating a set of possible memory states to every label in the program

- we have described **non relational analyses**

  $\mathcal{P}(\mathcal{V}\to\mathbb{Z})$ is abstracted into $\mathcal{V} \to \mathcal{P}(\mathbb{Z})$,
  and then $\mathcal{P}(\mathbb{Z})$ is abstracted into some $\mathcal{D}^\sharp$

- a **relational abstract domain** is some $\mathcal{D}^\sharp$ which is an abstraction of $\mathcal{P}(\mathcal{V}\to\mathbb{Z})$

  express that certain combinations of x and y are impossible

  (polyhedra, octagons)