Lambda Calculus

Meeting 16, CSCI 5535, Spring 2009

Announcements

- Homework 4 and 5 graded
  - Come talk to me
- Homework 6 due tonight
- No new homework this week. Work on your projects.
  - Come talk to me

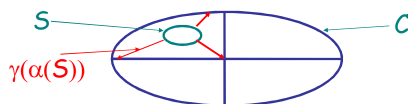
Review of Abstract Interpretation

Abstract Interpretation

- Abstract domain A (e.g., { -, 0, + })
- Concrete domain C (e.g., $\mathbb{Z}$)
- Abstraction function
  - $\alpha : \mathcal{P}(C) \to A$ or $\beta : C \to A$
- Concretization function
  - $\gamma : A \to \mathcal{P}(C)$
- Abstract semantics
  - $\sigma : \text{Exp} \to A$ (abstract operators $op^{\#}$)
- Concrete semantics
  - $[\![\cdot]\!] : \text{Exp} \to C$

Galois Connections

- A pair of functions between lattices A and $\mathcal{P}(C)$
  - γ and α are monotonic (with $\subseteq$ ordering on $\mathcal{P}(C)$)
  - $\alpha(\gamma(a)) = a$ for all $a \in A$
  - $\gamma(\alpha(S)) \supseteq S$ for all $S \in \mathcal{P}(C)$



Defining the Collecting Semantics

- We first define relations between the collecting semantics at different labels
  - We do it for unstructured CFGs (flowchart programs)
  - Can do it for IMP with careful notion of program points
- Define a label on each edge in the CFG
- For assignment

i
x := e
j

$C_j = \{\sigma[x := n] \mid \sigma \in C_i \wedge [\![e]\!]\sigma = n\}$

Abstract Interpretation on CFG

- Pick a complete lattice A (abstractions for $\mathcal{P}(\Sigma)$)
  - Along with a monotonic abstraction $\alpha : \mathcal{P}(\Sigma) \to A$
  - Alternatively, pick $\beta : \Sigma \to A$
  - This uniquely defines its Galois connection γ
- Take the relations between C_i and move them to the abstract domain:

  $a : \text{Label} \to A$
- Assignment

  Concrete: $C_j = \{\sigma[x := n] \mid \sigma \in C_i \wedge [\![e]\!]\sigma = n\}$

  Abstract: $a_j = \alpha\, \{\sigma[x := n] \mid \sigma \in \gamma(a_i) \wedge [\![e]\!]\sigma = n\}$

That's It! Program Analysis in a Nutshell

Define an Abstraction

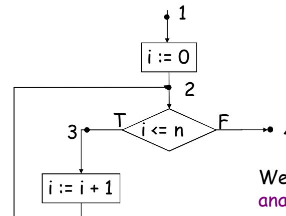
Compute a Fixed Point in the Abstraction

Other Abstract Domains

- Range analysis
  - Lattice of ranges: $R = \{\ \bot, [n..m], (-\infty, m], [n, +\infty), \top\ \}$
  - It is a complete lattice
    - $[n..m] \sqcup [n'..m'] = [\min(n, n')..\max(m,m')]$
    - $[n..m] \sqcap [n'..m'] = [\max(n, n')..\min(m, m')]$
    - With appropriate care in dealing with ∞
  - $\beta : \mathbb{Z} \to R$ such that $\beta(n) = [n..n]$
  - $\alpha : \mathcal{P}(\mathbb{Z}) \to R$ such that $\alpha(S) = \text{lub}\ \{\beta(n) \mid n \in S\} = [\min(S)..\max(S)]$
  - $\gamma : R \to \mathcal{P}(Z)$ such that $\gamma(r) = \{\ n \mid n \in r\ \}$
- This lattice has infinite-height chains
  - So the abstract interpretation might not terminate!

Example of Non-Termination

- Consider this (common) program fragment



What's this program?

We want to do range analysis for it

Example of Non-Termination

- Consider the sequence of abstract states at point 2
  - [1..1], [1..2], [1..3], …
  - The analysis never terminates
  - Or terminates very late if the loop bound known statically
- It is time to approximate even more: widening
- We redefine the join (lub) operator of the lattice to ensure that from [1..1] upon union with [2..2] the result is $[1..+\infty)$ and not [1..2]
- Now the sequence of states is
  - $[1..1], [1, +\infty), [1, +\infty)$ Done (no more infinite chains)

Formal Definition of Widening

- A widening $\triangledown : (P \times P) \to P$ on a poset $\langle P, \sqsubseteq \rangle$ satisfies:
  - $\forall\, x, y \in P\ .\ \ x \sqsubseteq (x \triangledown y) \ \ \wedge \ \ y \sqsubseteq (x \triangledown y)$

upper bound condition

What's this say?

termination condition

  - For all increasing chains

    $x^0 \sqsubseteq x^1 \sqsubseteq \ldots$

    the increasing chain

    $y^0 =^{def} x^0, \ldots, y^{n+1} =^{def} y^n \triangledown x^{n+1}, \ldots$

    is **not** strictly increasing.

Uses of Widening

- Two different main uses:
  - Approximate missing lubs. *(Not for us.)*
  - Convergence acceleration. *(This is the real use.)*
    - Used to compute an upper approximation of the least fixpoint of $F \in L \triangledown L$ starting from below when L does not satisfy the ascending chain condition.

The magic of program analysis

13

Formal Widening Example

$[1,1] \triangledown [1,2] = [1,+\infty)$

- Range Analysis on z:

L0: z := 1 ;
L1: while z<99 do
L2: z := z+1
L3: done /*z ≥ 99*/
L4:

$x^{Li}_j =^{def}$ the jth iterative attempt to compute an abstract value for z at label Li

Recall lub S = [min(S)..max(S)]
lub {[2,+∞),[1,+∞)} = {[1,+∞)}

Original x^i	Widened y^i
$x^{L0}_0 = \perp$	$y^{L0}_0 = \perp$
$x^{L1}_0 = [1,1]$	$y^{L1}_0 = [1,1]$
$x^{L2}_0 = [1,1]$	$y^{L2}_0 = [1,1]$
$x^{L3}_0 = [2,2]$	$y^{L3}_0 = [2,2]$
$x^{L2}_1 = [1,2]$	$y^{L2}_1 = [1,+\infty)$
$x^{L3}_1 = [2,+\infty)$	$y^{L3}_1 = [2,+\infty)$
$x^{L4}_0 = [99,+\infty)$	$y^{L4}_0 = [99,+\infty)$
stable (fewer than 99 iterations!)	

14

Other Abstract Domains

- Linear relationships between variables
  - A convex polyhedron is a subset of $\mathbb{Z}^k$ whose elements satisfy a number of inequalities:

    $a_1x_1 + a_2x_2 + \ldots + a_kx_k \geq c_i$
  - This is a complete lattice; linear programming methods compute lubs
- Linear relationships with at most two variables
  - Convex polyhedra but with ≤ 2 variables per constraint
  - Octagons ($x \pm y \geq c$) have efficient algorithms
- Modulus constraints (e.g. even and odd)

15

Abstract Chatter

- AI, Dataflow, and Software Model Checking
  - The big three (aside from flow-insensitive type systems) for program analyses
- Are in fact quite related:
  - David Schmidt. *Data flow analysis is model checking of abstract interpretation*. POPL '98.
- AI is usually flow-sensitive (per-label answer)
- AI can be path-sensitive (if your abstract domain includes ∨, for example), which is just where model checking uses BDD's
- Metal, SLAM, ESP, … can all be viewed as AI

16

Abstract Interpretation Summary

- AI is a very powerful technique that underlies a large number of program analyses
- AI can also be applied to functional and logic programming languages
- When the lattices have infinite height and widening heuristics are used, the result become harder to predictable
- AI is behind Astrée, which is used by Airbus

17

End of Abstract Interpretation

Questions?

Lambda Calculus

Lambda Calculus

Goal: Come up with a "core" language that's as small as possible and still Turing complete

This will give a way of illustrating important language features and algorithms

Plan

- Introduce lambda calculus
  - Syntax
  - Substitution
  - Operational Semantics
  - Evaluations strategies
  - Equality
- Later
  - Relationship to programming languages
  - Study of types and type systems

Lambda Background

- Developed in 1930's by Alonzo Church
- Subsequently studied by many people
  - Still studied today!
- Considered the "testbed" for procedural and functional languages
  - Simple
  - Powerful
  - Easy to extend with new features of interest
  - "Lambda:PL :: Turing Machine:Complexity"

"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus." (Landin '66)

Lambda Syntax

- The λ-calculus has 3 kinds of expressions (terms)

e ::=	x	Variables
|	λx. e	Functions (abstractions)
|	e_1 e_2	Application

- λx. e is a one-argument anonymous function with body e
- e_1 e_2 is a function application
- Application associates to the left

  x y z === (x y) z
- Abstraction extends as far to the right as possible

  λx. x λy. x y z === λx. (x [λy. {(x y) z}])

Why Should I Care?

- A language with 3 expressions? Woof!
- Li and Zdancewic. *Downgrading policies and relaxed noninterference*. POPL '05
  - Just one example of a recent PL/security paper

4. LOCAL DOWNGRADING POLICIES

4.1 Label Definition

Definition 4.1.1 (The policy language). *In Figure 1.*

Types	τ ::=	int | $\tau \to \tau$
Constants	c ::=	c_i
Operators	$\oplus$ ::=	$+, -, =, \ldots$
Terms	m ::=	$\lambda x{:}\tau.\ m \mid m\ m \mid x \mid c \mid m \oplus m$
Policies	n ::=	$\lambda x{:}\text{int}.\ m$
Labels	l ::=	$\{n_1, \ldots, n_k\} \quad (k \geq 1)$

Figure 1: L_{local} Label Syntax

The core of the policy language is a variant of the simply-typed λ-calculus with a base type, binary operators and con-

$$\frac{\Gamma \vdash m : \tau}{\Gamma \vdash m \equiv m : \tau} \quad \text{Q-REFL}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau}{\Gamma \vdash m_2 \equiv m_1 : \tau} \quad \text{Q-SYMM}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau \qquad \Gamma \vdash m_2 \equiv m_3 : \tau}{\Gamma \vdash m_1 \equiv m_3 : \tau} \quad \text{Q-TRANS}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash m_1 \equiv m_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ m_1 \equiv \lambda x{:}\tau_1.\ m_2 : \tau_1 \to \tau_2} \quad \text{Q-ABS}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau_1 \to \tau_2 \quad \Gamma \vdash m_3 \equiv m_4 : \tau_1}{\Gamma \vdash m_1\ m_3 \equiv m_2\ m_4 : \tau_2} \quad \text{Q-APP}$$

$\Gamma \vdash m_1 \equiv m_2 : \text{int}$

Examples of Lambda Expressions

- The identity function:

  $I =_{def} \lambda x.\ x$
- A function that, given an argument y, discards it and yields the identity function:

  $\lambda y.\ (\lambda x.\ x)$
- A function that, given an function f, invokes it on the identity function:

  $\lambda f.\ f\ (\lambda x.\ x)$

25

Examples of Lambda Expressions

- The identity function:

  $I =_{def} \lambda x.\ x$
- A function that, given an argument y, discards it and yields the identity function:

  $\lambda y.\ (\lambda x.\ x)$
- A function that, given an function f, invokes it on the identity function:

  $\lambda f.\ f\ (\lambda x.\ x)$

26

Scope of Variables

- As in all languages with variables, it is important to discuss the notion of scope
  - The scope of an identifier is the portion of a program where the identifier is accessible
- An abstraction $\lambda x.\ E$ binds variable x in E
  - x is the newly introduced variable
  - E is the scope of x (unless x is shadowed)
  - We say x is bound in $\lambda x.\ E$
  - Just like formal function arguments are bound in the function body

27

Free and Bound Variables

- A variable is said to be free in E if it has occurrences that are not bound in E
- We can define the free variables of an expression E recursively as follows:
  - $Free(x) = \{x\}$
  - $Free(E_1\ E_2) = Free(E_1) \cup Free(E_2)$
  - $Free(\lambda x.\ E) = Free(E) - \{x\}$
- Example: $Free(\lambda x.\ x\ (\lambda y.\ x\ y\ z)) = \{z\}$
- Free variables are (implicitly or explicitly) declared outside the expression

28

Free Your Mind!

- Just as in any language with statically-nested scoping we have to worry about variable shadowing
  - An occurrence of a variable might refer to different things in different contexts
- Example let-expressions (as in ML):

  let x = 5 in x + (let x = 9 in x) + x
- In λ-calculus:

  $\lambda x.\ x\ (\lambda x.\ x)\ x$

29

Renaming Bound Variables

= α-renaming or α-conversion

- λ-terms that can be obtained from one another by renaming bound variables are considered identical
- This is called α-equivalence
- Ex: $\lambda x.\ x$ is identical to $\lambda y.\ y$ and to $\lambda z.\ z$
- Intuition:
  - By changing the name of a formal argument and all of its occurrences in the function body, the behavior of the function *does not change*
  - In λ-calculus such functions are considered identical

30

Make It Easy On Yourself

- Convention: we will always try to rename bound variables so that they are all unique
  - e.g., write λx. x (λy.y) x instead of λx. x (λx.x) x
- This makes it easy to see the scope of bindings and also prevents confusion!

31

Substitution

- The substitution of e′ for x in e (written [e′/x]e)
  - Step 1. Rename bound variables in e and e′ so they are unique
  - Step 2. Perform the textual substitution of e′ for x in e
- Called capture-avoiding substitution

32

Substitution

- Example: [y (λx. x) / x] λy. (λx. x) y x
  - After renaming:

    [y (λx.x) / x] λz. (λu.u) z x
  - After substitution:

    λz. (λu.u) z (y (λx.x))

33

Substitution

- Example: [y (λx. x) / x] λy. (λx. x) y x
  - After renaming: [y (λx. x) / x] λz. (λu. u) z x
  - After substitution: λz. (λu. u) z (y (λx. x))
- If we are not careful with scopes we might get:

  λy. (λx. x) y (y (λx. x)) ← wrong!

34

Alternative: The de Bruijn Notation

- An alternative syntax that avoids naming of bound variables (and the subsequent confusions)
- The de Bruijn index of a variable *occurrence* is that number of lambda that separate the occurrence from its binding lambda in the abstract syntax tree
- The de Bruijn notation replaces names of occurrences with their De Bruijn indices
- Examples:
  - λ x. x — λ. 0
  - λ x. λ x. x — λ. λ. 0
  - λ x. λ y. y — λ. λ. 0
  - (λ x. x x) (λ z. z z) — (λ. 0 0) (λ. 0 0)
  - λ x. (λ x. λ y. x) x — λ. (λ. λ. 1) 0

Identical terms have identical representations!

35

Combinators

- A λ-term without free variables is closed or a combinator
- Some interesting combinators:

  I = λ x. x

  K = λ x. λ y. x

  S = λ f. λ g. λ x. f x (g x)

  D = λ x. x x

  Y = λ f. (λ x. f (x x)) (λ x. f (x x))

  Explain these informally
- Theorem: any closed term is equivalent to one written with just S, K and I
  - Example: D $=_\beta$ S I I
  - (we'll discuss this form of equivalence later)

36