Lecture 10. Advanced Topic

December 5, 2018

Outline

- Formal Semantics
- Program Analysis
- Julia, Rust, Stan

Advanced Formal Semantics and Program Analysis

Semantics

- How to generate values from various types of program constructs? e.g., Program constructs of ArithLang, VarLang, FuncLang, RefLang: expressions
- In this class, we have taught values and types
- (cover all cases – so we will work with grammar rules, because the grammar is the base to generate all strings)
- Programming language design: introduce a new language feature
  - what is/are the new value(s)?
  - what is/are the new type(s)
  - what is the new program construct (syntax)?
  - what is the semantic rules (value) for the new program construct?
  - what is the type rules for the new program construct?
  - Do we need to modify the semantic rules for old constructs?
  - Do we need to modify the type rules for old constructs?

Specifying Semantics

- Specifying value generation: Denotational, Axiomatic, Operational Semantics (how to generate values if the values are legitimate for each program construct)
- Specifying type rules (how to enforce rules on values – find and handle illegal values)
- Other rules for finding illegal values (e.g., type state rules)

Three Types of Semantics

Winskel 1993

- Operational semantics describes the meaning of a programming language by specifying how it executes on an abstract machine. We concentrate on the method advocated by Gordon Plotkin in his lectures at Aarhus on "structural operational semantics" in which evaluation and execution relations are specified by rules in a way directed by the syntax.
- Denotational semantics is a technique for defining the meaning of programming languages pioneered by Christopher Strachey and provided with a mathematical foundation by Dana Scott. At one time called "mathematical semantics," it uses the more abstract mathematical concepts of complete partial orders, continuous functions and least fixed points.
- Axiomatic semantics tries to fix the meaning of a programming contruct by giving proof rules for it within a program logic. The chief names associated with this approach are that of R.W.Floyd and C.A.R.Hoare. Thus axiomatic semantics emphasizes proof of correctness right from the start.

Logic

Semantics

Tony Hoare F-H logic

Robert Floyd

- *Denotational*: define the meaning of a program as elements of some abstract mathematical structure.
- *Axiomatic or Logical*: define the meaning of a program indirectly, by specifying program properties as assertions (and axioms) and using rules to establish the truths of those assertions. {P} c {Q} pre- Comma Post-
- *Operational*: define the meaning of a program in terms of the computation steps it takes in an idealised execution.

Denotational Semantics

`Number -> zero | (succ Number)`

Inductive semantics (Recall): buildup from some premitives

P(0) P(n)

- 0 is a number
- if n is a number then succ(n) is a number

$0, \mathtt{succ}(0), \mathtt{succ}(\mathtt{succ}(0)), \ldots$ are numbers

succ(0) is represented as 1

```
(define zero
  (lambda (f)
    (lambda (x)
      x)))
```

$[\![0]\!]$

$[\![\ ]\!] : P \to \mathbb{N}$

the set of program to the set of Numbers

Function

Using a list to represent unary representation

$$[\![0]\!] = \lambda f.\lambda x.x$$

$$[\![\mathbf{succ}(n)]\!] = \lambda n.\lambda f.\lambda x.(f\ ((n\ f)\ x))$$

$$[\![(+\ n_1\ n_2)]\!] = \lambda n_1.\lambda n_2.\lambda f.\lambda x.((((n_1\ succ)\ n_2)\ f)\ x)$$

$$[\![(-\ n_1\ n_2)]\!] = \ldots$$

Denotational Semantics

$[\![\ ]\!] : \textit{Expression} \to \lambda\text{-expression}$

$[\![\ ]\!] =$

Strategy to use

Write a function *eval*, which takes as argument a program in the language described by our grammar, and produces an integer following the semantics of the language. I.e.,

$$eval : \mathcal{P} \rightarrow \mathbb{N}$$

Operational Semantics

```
Program  ->  Expr
Expr     ->  Number | (+ Expr Expr) | (- Expr Expr) | (* Expr Expr)
```

Semantics of numbers and operations over numbers (basic expressions) are defined for a computing environment (e.g., Racket/Scheme).

- Use the semantics of numbers and operations from Racket
- Goal is to compute the semantics of compound exressions from the semantics of the basic expressions

$$eval(n) = \text{(numerical value of } n\text{)}$$

$$eval(+\ n_1\ n_2) = eval(n_1)\ + eval(n_2)$$

$$eval(-\ n_1\ n_2) = eval(n_1)\ - eval(n_2)$$

When we will use variables, we will also have the memory model for computing the semantics of the variables.

if-then-else

Extension to our grammar

```
Program    -> Expr
Expr       -> Number | ArithExpr | IfExpr
ArithExpr -> ...

IfExpr -> (CCond Expr Expr)
CCond  ->  BCond | (or CCond CCond) | (and CCond CCond)
         | (not CCond)
BCond  -> (gt Expr Expr) | (lt Expr Expr) | (eq Expr Expr)
```

Signatures

$eval : \mathcal{P} \rightarrow \mathbb{N}$

$evalcond : \mathcal{C} \rightarrow \{true, false\}$,
where $\mathcal{C}$ is the set of all conditional expressions

Semantics

Semantics of Conditional Expression

$$
\begin{aligned}
eval((CCond\ Expr_1\ Expr_2)) &= \text{if } evalcond(CCond) \text{ then} \\
&\quad eval(Expr_1) \text{ else } eval(Expr_2) \\
evalcond((gt\ Expr1\ Expr2)) &= eval(Expr1) > eval(Expr2) \\
evalcond((lt\ Expr1\ Expr2)) &= eval(Expr1) < eval(Expr2) \\
evalcond((eq\ Expr1\ Expr2)) &= eval(Expr1) == eval(Expr2) \\
evalcond((or\ CCond1\ CCond2)) &= evalcond(CCond1) \vee evalcond(CCond2) \\
evalcond((and\ CCond1\ CCond2)) &= evalcond(CCond1) \wedge evalcond(CCond2) \\
evalcond((not\ CCond)) &= \neg\ evalcond(CCond)
\end{aligned}
$$

Operational Semantics

1. Recursive Descent Parsing
2. Memory model/Environment ✓
   1. Environment mimics the scope of the operations
   2. Environment must have get and set methods

Notations

- $[\![\texttt{Program}]\!]_\sigma$: Semantics of the `Program` in the context of the environment σ

Set: (*Variable Assignment*) $\circ\, \sigma$
add the tuple at the left-most position in σ (head of a list)

Get: σ(*Variable*)
value of the left-most occurrence of variable in σ

Typical Operational Semantics as Rules of Inferences

$$\frac{\texttt{Antecedent}}{\texttt{Consequent}}$$

For instance,

$$\frac{[\![E_1]\!]_\sigma \downarrow n_1 \text{ and } [\![E_2]\!]_\sigma \downarrow n_2}{[\![(+\ E_1\ E_2)]\!]_\sigma \downarrow\ n_1\ +\ n_2}$$

or,

$$[\![(+\ E_1\ E_2)]\!]_\sigma \downarrow n_1\ +\ n_2 \Leftarrow [\![E_1]\!]_\sigma \downarrow n_1 \text{ and } [\![E_2]\!]_\sigma \downarrow n_2$$

or,

$$[\![(+\ E_1\ E_2)]\!]_\sigma = [\![E_1]\!]_\sigma + [\![E_2]\!]_\sigma$$

Operational Semantics $[\![]\!] : \mathcal{E} \times \Sigma \to \mathbb{N}$

Let $\mathcal{E}$: set of all possible expressions; Σ: set of all possible environments;
$\mathbb{N}$: set of all numbers

eval (number) = Number

1. $[\![\texttt{Number}]\!]_\sigma = \texttt{Number}$

2. $[\![\texttt{Variable}]\!]_\sigma = \sigma(\texttt{Variable})$

3. $[\![\texttt{(+ Expr1 Expr2)}]\!]_\sigma =$ sum of $[\![\texttt{Expr1}]\!]_\sigma$ and $[\![\texttt{Expr2}]\!]_\sigma$

4. $$[\![\texttt{(CCond Expr1 Expr2)}]\!]_\sigma = \begin{cases} [\![\texttt{Expr1}]\!]_\sigma & \text{if } ((\texttt{CCond}))_\sigma \text{ is true} \\ [\![\texttt{Expr2}]\!]_\sigma & \text{otherwise} \end{cases}$$

5. $[\![\texttt{(var (Variable Expr1) Expr2)}]\!]_\sigma = [\![\texttt{Expr2}]\!]_{(\texttt{Variable } [\![\texttt{Expr1}]\!]_\sigma) \circ \sigma}$

let

$\circ \sigma$

Operational Semantics $(\!(\,)\!) : \mathcal{B} \times \Sigma \to \{true, false\}$

$[\![\,]\!]$

Let $\mathcal{B}$: set of all possible boolean expressions; Σ: set of all possible environments

1. $(\!(\texttt{(gt Expr1 Expr2)})\!)_\sigma = [\![\texttt{Expr1}]\!]_\sigma > [\![\texttt{Expr2}]\!]_\sigma$
2. $(\!(\texttt{(lt Expr1 Expr2)})\!)_\sigma = [\![\texttt{Expr1}]\!]_\sigma < [\![\texttt{Expr2}]\!]_\sigma$
3. $(\!(\texttt{(eq Expr1 Expr2)})\!)_\sigma = [\![\texttt{Expr1}]\!]_\sigma == [\![\texttt{Expr2}]\!]_\sigma$
4. $(\!(\texttt{(or CCond1 CCond2)})\!)_\sigma = (\!(\texttt{CCond1})\!)_\sigma \vee (\!(\texttt{CCond2})\!)_\sigma$
5. $(\!(\texttt{(not CCond)})\!)_\sigma = \neg\, (\!(\texttt{CCond})\!)_\sigma$

Examples

Semantics require an initial environment Let $\sigma = \epsilon$

$$
\begin{aligned}
& [\![\texttt{(var (x 1) ((gt x 0) (+ x 1) (+ x 2)))}]\!]_\sigma \\
= \; & [\![\texttt{((gt x 0) (+ x 1) (+ x 2))}]\!]_{\texttt{(x } [\![\texttt{1}]\!]_\sigma\texttt{)} \circ \sigma} \\
= \; & [\![\texttt{((gt x 0) (+ x 1) (+ x 2))}]\!]_{\texttt{(x 1)}} \\
= \; & \begin{cases} [\![\texttt{(+ x 1)}]\!]_{\texttt{(x 1)}} & \text{if } (\!(\texttt{(gt x 0)})\!)_{\texttt{(x 1)}} \\ [\![\texttt{(+ x 2)}]\!]_{\texttt{(x 1)}} & \text{otherwise} \end{cases} \\
= \; & \begin{cases} [\![\texttt{(+ x 1)}]\!]_{\texttt{(x 1)}} & \text{if } [\![\texttt{x}]\!]_{\texttt{(x 1)}} > [\![\texttt{0}]\!]_{\texttt{(x 1)}} \\ [\![\texttt{(+ x 2)}]\!]_{\texttt{(x 1)}} & \text{otherwise} \end{cases} \\
= \; & [\![\texttt{(+ x 1)}]\!]_{\texttt{(x 1)}} \\
= \; & \ldots
\end{aligned}
$$

Examples

Semantics require an initial environment Let $\sigma = \epsilon$

$$
\begin{aligned}
[\![\texttt{(var (x 1) (+ x 4))}]\!]_\sigma &= [\![\texttt{(+ x 4)}]\!]_{(\texttt{x } [\![\texttt{1}]\!]_\sigma) \circ \sigma} \\
&= [\![\texttt{(+ x 4)}]\!]_{\texttt{(x 1)}} \\
&= \text{add } [\![\texttt{x}]\!]_{\texttt{(x 1)}} \text{ to } [\![\texttt{4}]\!]_{\texttt{(x 1)}} \\
&= \text{add } 1 \text{ and } 4 \\
&= 5
\end{aligned}
$$

$$
\begin{aligned}
*[\![\texttt{(var (x 1) (var (x (+ x 1) (+ x 4))))}]\!]_\sigma &= \\
[\![\texttt{(var (x (+ x 1) (+ x 4)))}]\!]_{(\texttt{x } [\![\texttt{1}]\!]_\sigma) \circ \sigma} &= \\
[\![\texttt{(var (x (+ x 1) (+ x 4)))}]\!]_{\texttt{(x 1)}} &= \\
[\![\texttt{(+ x 4)}]\!]_{(\texttt{x } [\![\texttt{(+ x 1)}]\!]_{\texttt{(x 1)}}) \circ \texttt{(x 1)}} &= \\
[\![\texttt{(+ x 4)}]\!]_{\texttt{(x 2)} \circ \texttt{(x 1)}} &= \\
\text{add } [\![\texttt{x}]\!]_{\texttt{(x 2)} \circ \texttt{(x 1)}} \text{ and } [\![\texttt{4}]\!]_{\texttt{(x 2)} \circ \texttt{(x 1)}} &= \\
\ldots &
\end{aligned}
$$

Some More Theory

- The rules of operational semantics are often called *evaluation rules* or *inference rules*.
- In general, we have an evaluation/inference rule for each language construct.
- Operational semantics can be viewed as a proof system (used to prove or disprove the valuation of a valid sentence in the language of the corresponding grammar)
- A proof system is Complete if every true judgment is provable, i.e., if one makes the *true judgment* that a program evaluates to v, then there is a sequence of evaluation rules that can be applied to the program to obtain v.
- A proof system is Sound if every provable judgment is true, i.e., if there is a sequence of evaluation rules that evaluates a program to obtain v, then the judgment that the program evaluates to v is true.

Completeness and Soundness

If the only evaluation rule for addition is
$[\![(+\ \texttt{Expr1}\ \texttt{Expr2})]\!]_\sigma = [\![\texttt{Expr1}]\!]_\sigma$ if $[\![\texttt{Expr2}]\!]_\sigma == 0$
then our proof system is incomplete but sound.

If the evaluation rule for addition is
$[\![(+\ \texttt{Expr1}\ \texttt{Expr2})]\!]_\sigma = [\![\texttt{Expr1}]\!]_\sigma$
then our proof system is unsound and incomplete

Completeness and Soundness

Correctness of implementation

Every semantic rule has been implemented

The implementation correctly realizes the semantic rule

Collectively ensures correctness

Axiomatic Semantics

```
Program  ->  Expr
Expr     ->  Number | (+ Expr Expr) | (- Expr Expr) | (* Expr Expr)
```

Describe the pre- and post-conditions of the program statements as logical statements.

$$\frac{n_1 \text{ is a number} \geq 0 \ \wedge \ n_2 \text{ is a number} \geq 0}{(+\ n_1\ n_2)}$$

$$n_1 \leq (+\ n_1\ n_2) \ \wedge \ n_2 \leq (+\ n_1\ n_2)$$

pre-

post-

So $\models \{A\}c\{B\}$ iff for all interpretations I, if c is executed from a state which satisfies A then if its execution terminates in a state that state will satisfy B.[1] □

Exercise 6.7 In an earlier exercise it was asked to write down an assertion $A \in \mathbf{Assn}$ with one free integer variable i expressing that i was prime. By working through the appropriate cases in the definition of the satisfaction relation $\models^I$ between states and assertions, trace out the argument that $\models^I A$ iff $I(i)$ is indeed a prime number. □

6.4 Proof rules for partial correctness

We present proof rules which generate the valid partial correctness assertions. The proof rules are syntax-directed; the rules reduce proving a partial correctness assertion of a compound command to proving partial correctness assertions of its immediate subcommands. The proof rules are often called *Hoare rules* and the proof system, consisting of the collection of rules, *Hoare logic*.

Rule for **skip**:

$$\{A\}\mathbf{skip}\{A\}$$

Rule for assignments:

$$\frac{}{\{B[a/X]\}X := a\{B\}}$$

Whereve X, replace with a $[a/X]$

Rule for sequencing:

$$\frac{\{A\}c_0\{C\} \quad \{C\}c_1\{B\}}{\{A\}c_0; c_1\{B\}}$$

Rule for conditionals:

$$\frac{\{A \wedge b\}c_0\{B\} \quad \{A \wedge \neg b\}c_1\{B\}}{\{A\}\mathbf{if}\ b\ \mathbf{then}\ c_0\ \mathbf{else}\ c_1\{B\}}$$

Rule for while loops:

$$\frac{\{A \wedge b\}c\{A\}}{\{A\}\mathbf{while}\ b\ \mathbf{do}\ c\{A \wedge \neg b\}}$$

Rule of consequence:

$$\frac{\models (A \Rightarrow A') \quad \{A'\}c\{B'\} \quad \models (B' \Rightarrow B)}{\{A\}c\{B\}}$$

[1] The picture suggests, incorrectly, that the extensions of assertions A^I and B^I are disjoint; they will both always contain $\perp$, and perhaps have other states in common.

Axiomatic

where A and B are assertions like those we've already seen in **Bexp** and c is a command. The precise interpretation of such a compound assertion is this:

> for all states σ which satisfy A if the execution c from state σ terminates in state σ' then σ' satisfies B.

Put another way, $\{A\}c\{B\}$ means that any successful (*i.e.*, terminating) execution of c from a state satisfying A ends up in a state satisfying B. The assertion A is called the precondition and B the postcondition of the partial correctness assertion $\{A\}c\{B\}$.

Assertions of the form $\{A\}c\{B\}$ are called *partial correctness assertions* because they say nothing about the command c if it fails to terminate. As an extreme example consider

$$c \equiv \textbf{while true do skip}.$$

The execution of c from any state does not terminate. According to the interpretation we give above the following partial correctness assertion is valid:

$$\{\textbf{true}\}c\{\textbf{false}\}$$

simply because the execution of c does not terminate. More generally, because c loops, any partial correctness assertion $\{A\}c\{B\}$ is valid. Contrast this with another notion, that of total correctness. Sometimes people write

$$[A]c[B]$$

to mean that the execution of c from any state which satisfies A will terminate in a state which satisfies B. In this book we shall not be concerned much with total correctness assertions.

Warning: There are several different notations around for expressing partial and total correctness. When dipping into a book make doubly sure which notation is used there.

We have left several loose ends. For one, what kinds of assertions A and B do we allow in partial correctness assertions $\{A\}c\{B\}$? We say more in a moment, and turn to a more general issue.

The next issue can be regarded pragmatically as one of notation, though it can be viewed more conceptually as the semantics of assertions for partial correctness—see the "optional" Section 7.5 on denotational semantics using predicate transformers. Firstly let's introduce an abbreviation to mean the state σ satisfies assertion A, or equivalently the assertion A is true at state σ. We abbreviate this to:

$$\sigma \models A.$$

Program Analysis - Goals

Predict program behavior without running all inputs:

- Is there possibly an erroneous state?
- Which inputs can exercise changes
- Are the two inputs going to exercise the same program paths? if so, we just need to run one input
- Do the two statements produce the same values?
- ...

Program Analysis - Applications

- Assist compiler optimizations (accelerate computation)
- Detect vulnerabilities and bugs
- Automatically generate test inputs
- Summarize the code and infer specifications
- ...

Relations with Programming Languages

- Analyze the program written in the language
- Define a subset of language constructs
- For each type of program construct, what information you collect
  - control flow information: what is the next statement to execute?
  - dependency analysis: does this statement possibly affect the results at the end of program?
  - symbolic analysis: what are the symbolic output for the statement

Relations with Programming Languages

- Operational semantics, also called concrete semantics: rules to compute each value at program points, computing values for one input
- Program analysis computes properties – abstract semantics: rules to compute a condition for possible program states at a program point, reasoning program properties for all inputs or a subset of inputs

Julia by MIT

- combining features of imperative, functional, and object-oriented programming.
- for high-level numerical computing
- Parallel running back end
- Easy to call c and fortune library
- Language features: arithmetic, function, variable, control flow, multi-dimensional array
- Dynamic programming language (lisp, ruby): dynamically-typed, feels like a scripting language
- Julia Notebook and examples

Rust by Firefox:
https://doc.rust-lang.org/rust-by-example/

- guaranteed memory safety
- pattern matching
- type inference
- minimal runtime
- efficient C bindings
- ...

Stan: http://mc-stan.org/

- a probabilistic programming language for statistical inference written in C++
- specify a (Bayesian) statistical model with an imperative program calculating the log probability density function

9. Regression Models

Stan supports regression models from simple linear regressions to multilevel generalized linear models.

9.1. Linear Regression

The simplest linear regression model is the following, with a single predictor and a slope and intercept coefficient, and normally distributed noise. This model can be written using standard regression notation as

$$y_n = \alpha + \beta x_n + \epsilon_n \quad \text{where} \quad \epsilon_n \sim \text{Normal}(0, \sigma).$$

This is equivalent to the following sampling involving the residual,

$$y_n - (\alpha + \beta X_n) \sim \text{Normal}(0, \sigma),$$

and reducing still further, to

$$y_n \sim \text{Normal}(\alpha + \beta X_n, \sigma).$$

This latter form of the model is coded in Stan as follows.

```
data {
  int<lower=0> N;
  vector[N] x;
  vector[N] y;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  y ~ normal(alpha + beta * x, sigma);
}
```

There are `N` observations, each with predictor `x[n]` and outcome `y[n]`. The intercept and slope parameters are `alpha` and `beta`. The model assumes a normally distributed noise term with scale `sigma`. This model has improper priors for the two regression coefficients.

```
data {
  int<lower=2> K; // num topics
  int<lower=2> V; // num words
  int<lower=2> U; // num users
  int<lower=2> I; // num items
  int<lower=1> N; // total word instances
  int<lower=1,upper=V> word[N]; // word n
  int<lower=1,upper=I> item[N]; // item ID for word n
  int<lower=1,upper=U> user[N]; // user ID for word n
  vector<lower=0, upper=1>[K] alpha_user; // topic
      prior concentrations for users
  vector<lower=0, upper=1>[K] alpha_item; // topic
      prior concentrations for items
  vector<lower=0, upper=1>[V] beta; // prior
      probability for seeing word
}
parameters {
  simplex[K] item_topics[I]; // topic dist for item i
  simplex[K] user_topics[U]; // topic dist for user u
  simplex[V] word_topics[K]; // for topic k prob of
      seeing word v
}
model {
  for (i in 1:I)
    item_topics[i] ~ dirichlet(alpha_item); // prior
        on item topics
  for (u in 1:U)
    user_topics[u] ~ dirichlet(alpha_user); // prior
        on user topics
  for (k in 1:K)
    word_topics[k] ~ dirichlet(beta); // prior
  // for every word in our corpus
  for (n in 1:N) {
    real gamma[K];
    // for every topic
    for (k in 1:K){
      // topic distribution for this user
      gamma[k] <- log(item_topics[item[n], k] +
          user_topics[user[n], k]) + log(word_topics[k
          , word[n]]);
    }
    increment_log_prob(log_sum_exp(gamma)); //
        likelihood
  }
}
```

Fig. 1: Example – Original Code

```
data {
  int<lower=2> K;
  int<lower=2> V;
  vector<lower=0, upper=1>[V]
      beta;
}
parameters {
  simplex[V] word_topics[K];
}
model {
  for (k in 1:K)
    word_topics[k] ~ dirichlet(
        beta);
}
```

Fig. 2: Example – Reduced code

Fig. 3: Example – Code and Data Reduction

Reduction	%	Ratio
Lines of Code	64%	(12/33)
Code Constructs	53%	(14/30)
Data Points	98%	(74B/3KB)
Algorithm Iters.	0%	(100/100)

formation was performed. This result clearly demonstrates PReduce's ability in reducing probabilistic programs using a efficient mix of basic and domain specific transformations.

III. Framework

In this section, we discuss the framework for PReduce. Figure 4 shows the high-level architecture of PReduce. Now, we discuss each component in details. The input to PReduce is a set of buggy probabilistic program artifacts, each with the buggy program, data and a script which runs the program (reduced) and checks for the existence of the expected bug. PReduce takes each buggy program, parses it and builds a parse tree (Section III-A). It then tries to apply various basic and domain-specific source-to-source transformations. PReduce ensures that each generated program is syntactically valid (Section III-B). PReduce follows an iterative algorithm for applying the transformations to reduce the program, data and the number of samples. Finally, PReduce compares the reduced program to the original program and computes the metrics to evaluate the quality of reduction (Section III-D).

A. Parser

PReduce takes as input an existing Stan program. To transform this program, we used used Antlr [23] to build a parser for Stan. In this work, we defined a subset of Stan's grammar that was sufficient to parse all of the programs we extracted from the bug reports. The parser automatically builds the parse tree of the program. Antlr also provides a visitor interface that can be used to walk through the parse tree and make various kinds of modifications on top of the tree.

B. Transformer

The transformer is responsible for applying transformations on the input program that simplify its structure. After each transformation, the new program should preserve the same

Follow-up Courses

- COM S 440, Principles and Practices of Compiling: parsing algorithms, compiler optimizations, code generation
- COM S 413x/513x, Foundations and Applications of Program Analysis: applications and analysis behind them

Program analysis research group topics:

If interested, email me @ weile@iastate.edu

1. finding bugs (control flow analysis, data flow analysis, pointer analysis, interprocedual analysis, demand-driven analysis)
2. inferring specifications, assertions (information flow, source/sink problems, pre-/post-conditions, finite state machine construction and trace analysis, program invariant inference)
3. automatically generating test inputs (fuzzing, symbolic execution)
4. debugging (dependency analysis, statistical analysis with program analysis)
5. analyzing changes and versions (incremental analysis, history analysis, differential analysis)
6. big code analysis (machine learning, nlp, collective program analysis)
7. certifying and explaining AI software (abstract interpretation)

- COM S 540, Advanced Programming Languages: type systems