Fuzz Testing Using Constraint Logic Programming

Kyle Dewey, Jared Roesch, Ben Hardekopf

"Language Fuzzing Using Constraint Logic Programming" in ASE'14

Fuzz Testing

- Automatic generation of program inputs for testing and confidence-building
- In this work, we focus specifically on testing **compilers and interpreters**
  - Programs are inputs
  - Multiple implementations available

State of the Art: Stochastic Grammars

First take a grammar...

$$e \in \mathit{ArithExp} ::= n \in \mathbb{N} \;\mid\; e_1 + e_2$$

State of the Art: Stochastic Grammars

First take a grammar...

$$\boxed{e \in \mathit{ArithExp}} ::= n \in \mathbb{N} \mid e_1 + e_2$$

State of the Art: Stochastic Grammars

First take a grammar...

$$e \in \mathit{ArithExp} ::= \boxed{n \in \mathbb{N}} \;\mid\; e_1 + e_2$$

State of the Art: Stochastic Grammars

First take a grammar...

$$e \in ArithExp ::= n \in \mathbb{N} \ \mid \ \boxed{e_1 + e_2}$$

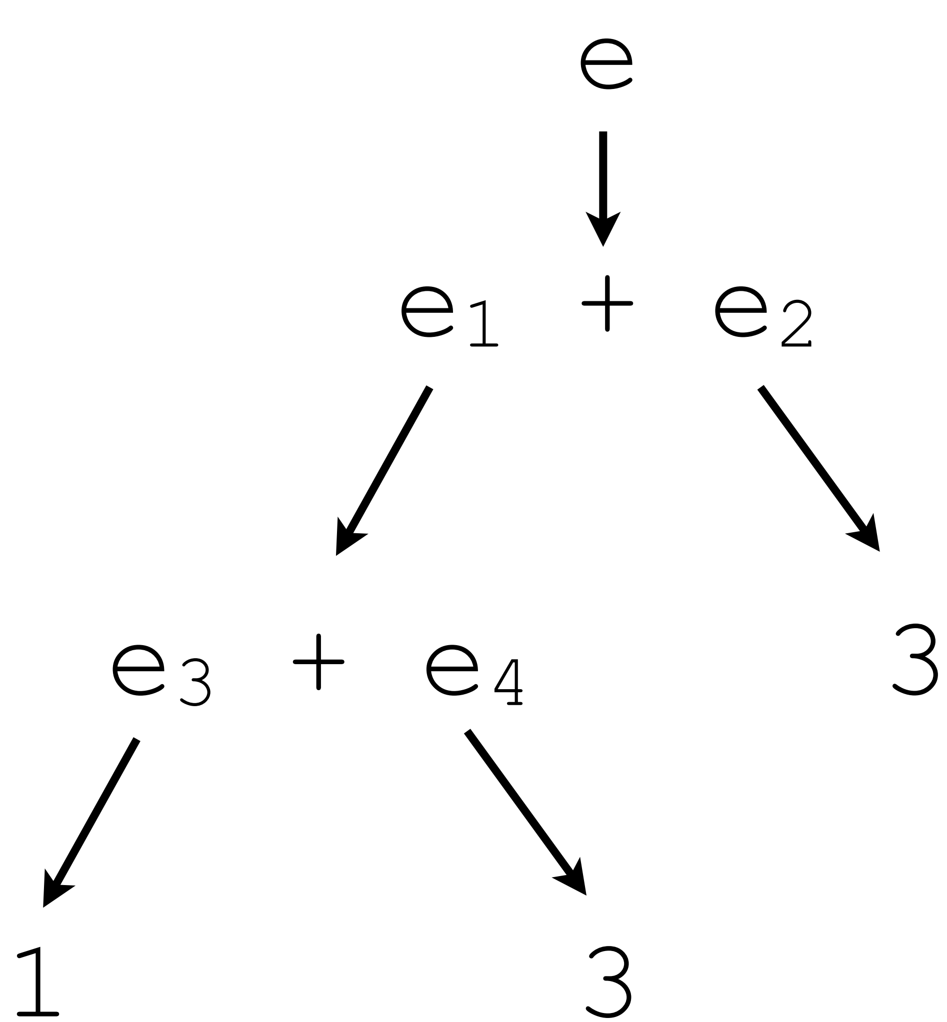
State of the Art: Stochastic Grammars

...then annotate with probabilities associated with the likelihood of generating a particular production

$$e \in ArithExp ::= n \in \mathbb{N}^{0.6} \mid e_1 + e_2^{0.4}$$

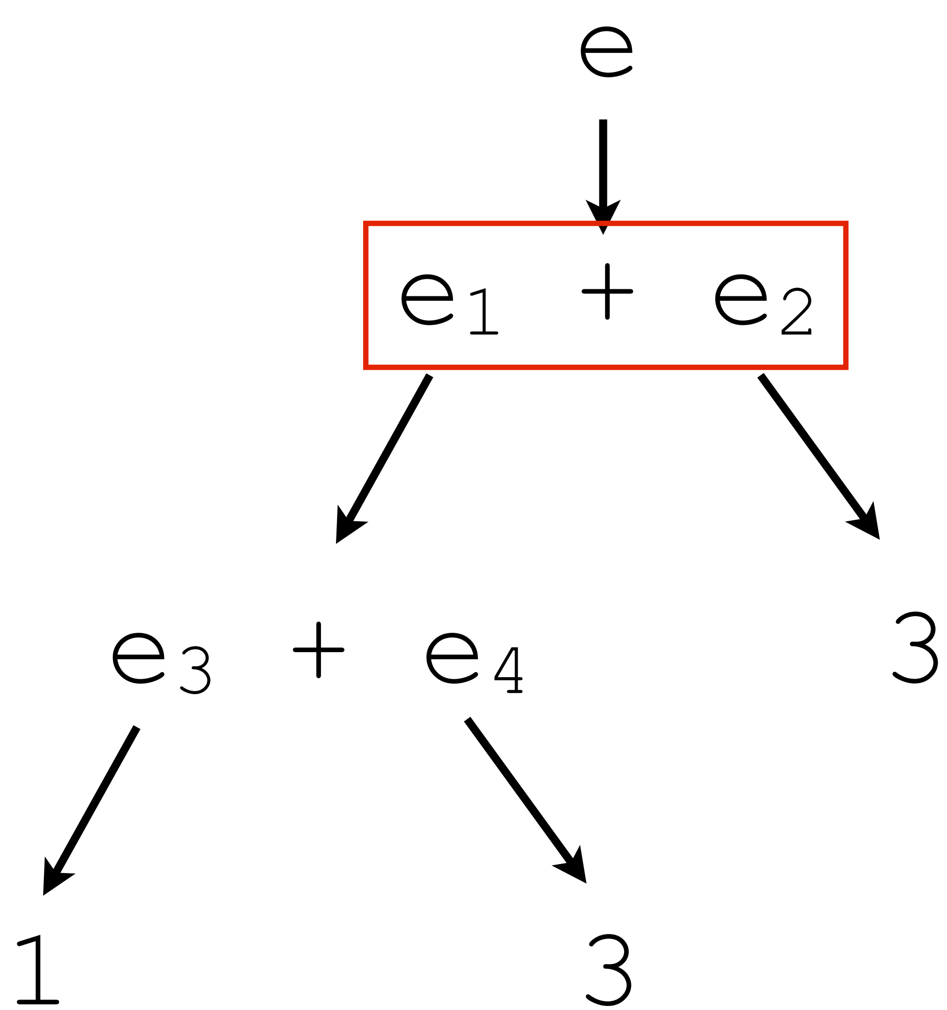
Example Derivation

$$e \in \mathit{ArithExp} ::= \overset{0.6}{n \in \mathbb{N}} \;\Big|\; \overset{0.4}{e_1 + e_2}$$



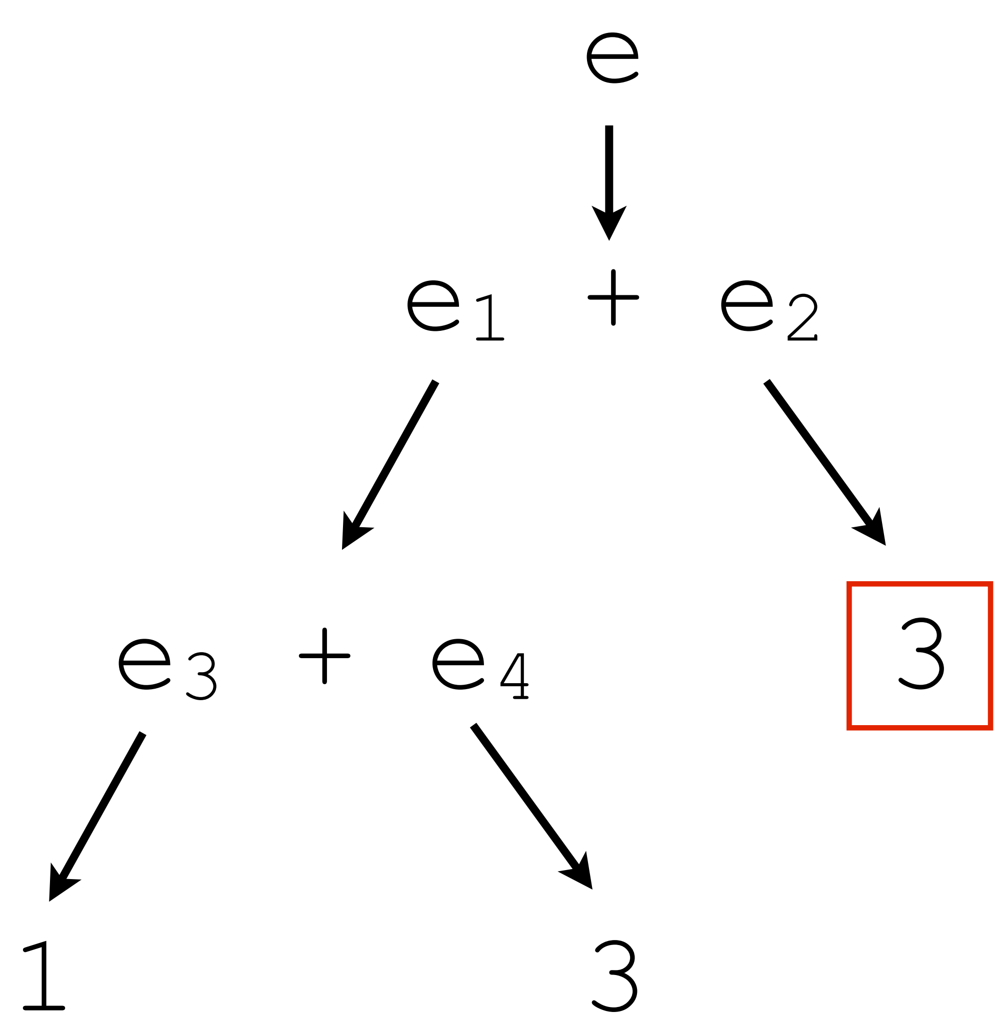
Example Derivation

$$e \in \mathit{ArithExp} ::= n \in \mathbb{N}^{\;0.6} \mid \boxed{e_1 + e_2^{\;0.4}}$$



Example Derivation

$$e \in \mathit{ArithExp} ::= n \in \mathbb{N}^{\,0.6} \mid e_1 + e_2^{\,0.4}$$



Stochastic Weaknesses

- Difficult to focus in on anything beyond simple syntactic properties
  - Well-typed programs
  - Expressions that evaluate to some value
- Probabilities only allow for very coarse-grained configuration
- Hard to increase confidence in specific implementation components

Enter Constraint Logic Programming (CLP)

- Allows for the specification of relational and arithmetic constraints on symbolic variables
- Can easily encode grammars
- Can specify generators focusing in on both syntactic and semantic program properties
- **Generalizes** stochastic grammars

Encoding the Grammar

$$e \in \mathit{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$$

Encoding the Grammar

$$e \in \mathit{ArithExp} ::= n \in \mathbb{N} \;\mid\; e_1 + e_2$$

Encoding the Grammar

$$e \in \mathit{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$$

```
arithExp(num(N)) :-
   INTMIN #=< N,
   N #=< INTMAX.
```

Encoding the Grammar

$$e \in \mathit{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$$

```
arithExp(num(N)) :-
   INTMIN #=< N,
   N #=< INTMAX.
```

Encoding the Grammar

$$e \in \mathit{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2$$

```
arithExp(num(N)) :-
   INTMIN #=< N,
   N #=< INTMAX.

arithExp(add(E1, E2)) :-
   arithExp(E1),
   arithExp(E2).
```

Making it Stochastic

$$e \in \mathit{ArithExp} ::= \overset{0.6}{n \in \mathbb{N}} \;\mid\; \overset{0.4}{e_1 + e_2}$$

```
arithExp(num(N)) :-
   INTMIN #=< N,
   N #=< INTMAX.

arithExp(add(E1, E2)) :-
   arithExp(E1),
   arithExp(E2).
```

Making it Stochastic

$$e \in \mathit{ArithExp} ::= n \in \mathbb{N}^{\,0.6} \;\middle|\; e_1 + e_2^{\,0.4}$$

```
arithExp(num(N)) :-
   maybe(0.6),
   INTMIN #=< N,
   N #=< INTMAX.

arithExp(add(E1, E2)) :-
   arithExp(E1),
   arithExp(E2).
```

Generation

With the query:

```
:- arithExp(E), writeln(E), fail.
```

...`E` is nondeterministically bound to all productions of the grammar.

Example: Expressions that Evaluate to 7

```
eval(num(N), N).
eval(add(E1, E2), N) :-
  eval(E1, N1),
  eval(E2, N2),
  N #= N1 + N2.

% same arithExp from before
evalsTo7(E) :-
  arithExp(E),
  eval(E, 7).
```

Application

- Applied to both JavaScript (dynamically typed) and Scala (statically typed)
- Challenge with JavaScript: generating programs which do specific things
- Challenge with Scala: generating well-typed programs

Application to JavaScript

- Four generators developed that make four different kinds of programs:
  - `js-err`: Programs that avoid runtime type errors
  - `js-overflow`: Programs that overflow
  - `js-inher`: Programs that use prototype-based inheritance
  - `js-withclo`: Programs that intermix JavaScript's `with` and closures in specific ways

Application to Scala

- Two generators for two type systems:
  - `scala-base`: Simply-typed core that covers function and method calls
  - `scala-full`: Adds `match` (pattern matching), generics, subtyping, and inheritance

Evaluation

- Interested in measuring the rate at which these generators can generate **programs of interest** relative to stochastic techniques, along with their **added complexity** over stochastic techniques
- Hypothesis: these custom generators can generate interesting programs at a much faster rate than stochastic techniques, without much additional complexity

Generation Results

	In programs per second		
Generator	**Stochastic-based**	**CLP-based**	**CLP / Stochastic**
js-err	9,880	37,759	3.8
js-overflow	123	958	7.8
js-inher	0	126,194	∞
js-withclo	0.04	125,901	3,147,525
scala-base	56	105,510	1,884
scala-full	0	183,187	∞

Added Complexity Results

Generator	**LOC**
`js-stoc`	340
`js-err`	429
`js-overflow`	360
`js-inher`	397
`js-withclo`	394
`scala-stoc-base`	109
`scala-base`	106
`scala-stoc-full`	167
`scala-full`	245

See ASE'14 Paper for Details...

- Alternate search strategies
- More on different type systems
- Embedded CLP DSLs for fuzzing
- Total and unique stochastic programs generated

Conclusions

- Stochastic grammars generally cannot focus in on the generation of specific programs
- Our CLP-based approach generalizes stochastic grammars, allowing for targeted generation without much additional code