Language Fuzzing Using Constraint Logic Programming

Kyle Dewey, Jared Roesch, Ben Hardekopf

Language Fuzzing

- Automatic program generation technique for testing compilers and interpreters
- Can be used to build confidence in a whole implementation or in parts of an implementation

State of the Art: Stochastic Grammars

First take a grammar...

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

State of the Art: Stochastic Grammars

First take a grammar...

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

State of the Art: Stochastic Grammars

First take a grammar...

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

State of the Art: Stochastic Grammars

First take a grammar...

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

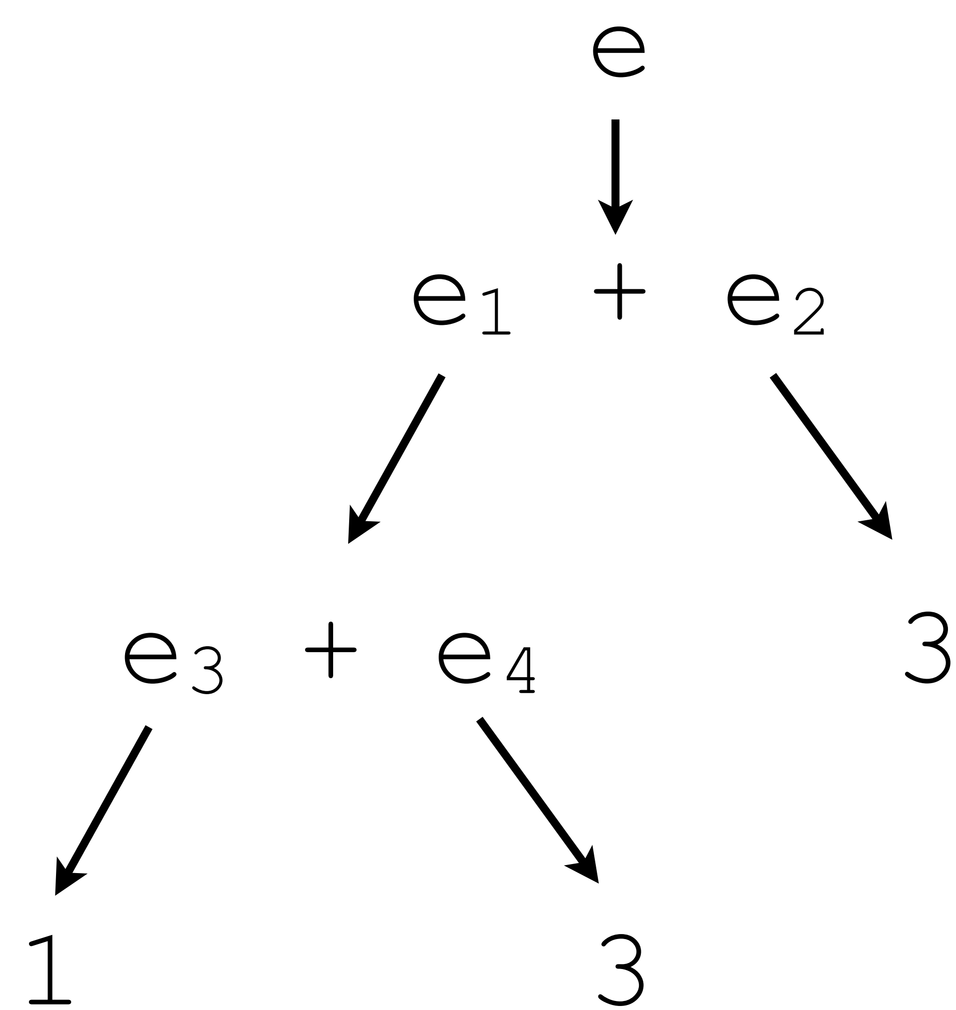
State of the Art: Stochastic Grammars

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

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

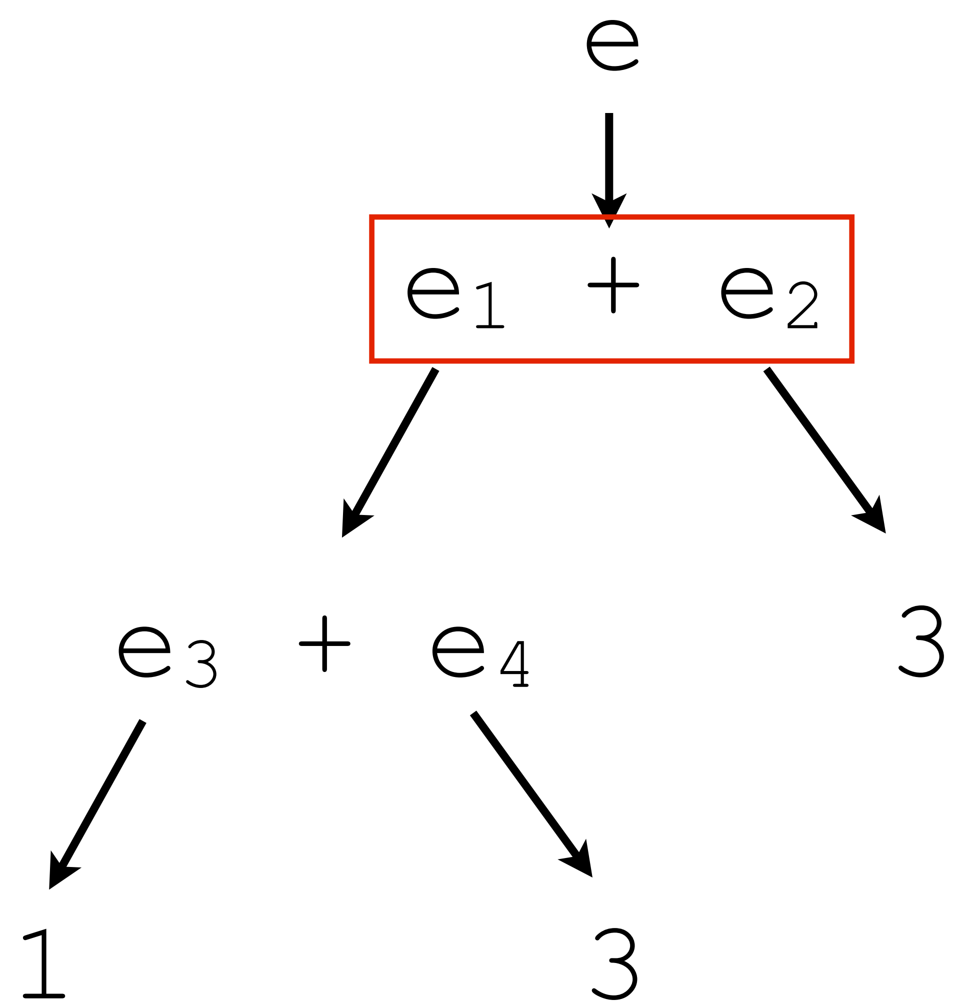
Example Derivation

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



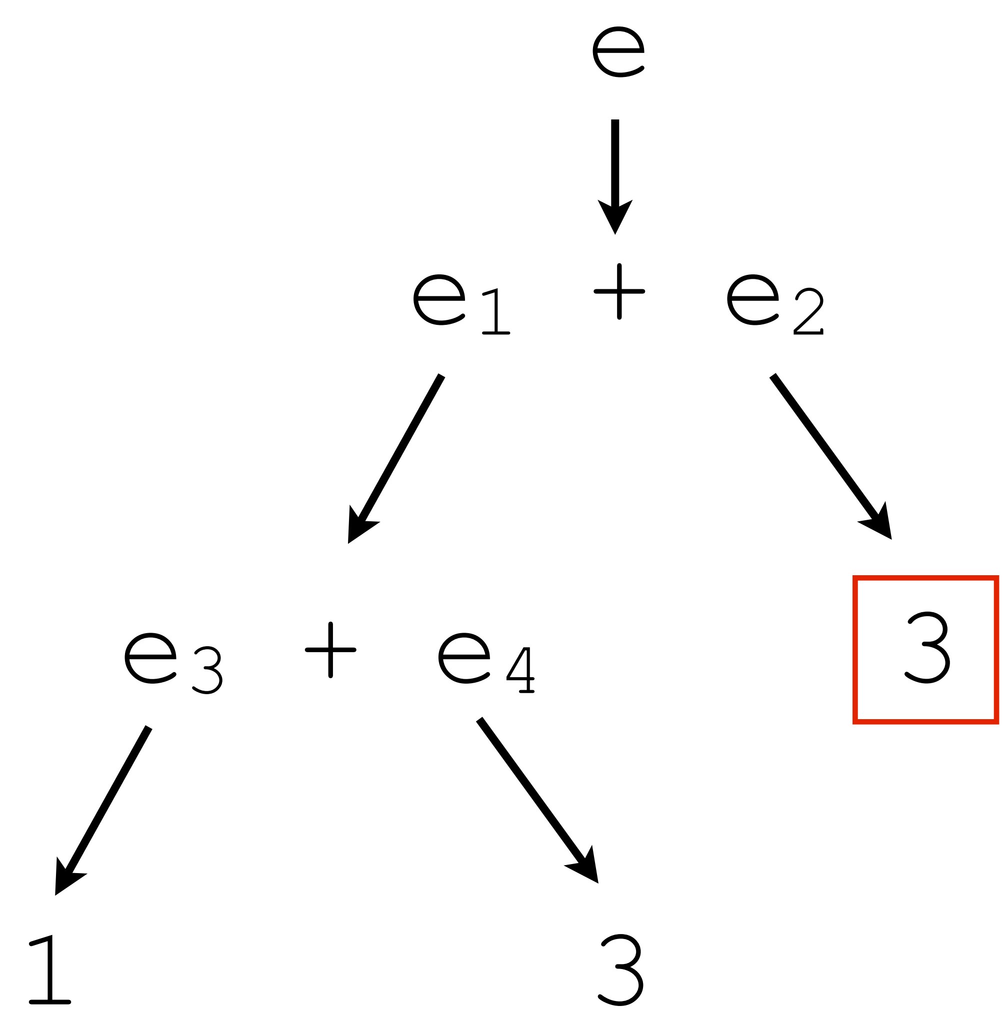
Example Derivation

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



Example Derivation

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



Stochastic Weaknesses

- Difficult to focus in programs that do specific things (e.g., expressions that evaluate to 7)
- 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

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

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

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

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

Encoding the Grammar

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

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

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

Making it Stochastic

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

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

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

Making it Stochastic

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

```
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.

Generalization: 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 generating JavaScript programs
- 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

Evaluation

- Interested in measuring the rate at which these generators can generate **programs of interest** relative to stochastic techniques
- Hypothesis: these custom generators can generate interesting programs at a much faster rate than stochastic techniques

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

See Paper for Details...

- Alternate search strategies
- 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