Targeted Automated Testing Using Constraint Logic Programming

Kyle Dewey
Advisor: Ben Hardekopf
Targeted Automated Testing Using Constraint Logic Programming

Goal is to test some piece of software in the hopes of finding bugs before users do
Targeted Automated Testing Using Constraint Logic Programming

Automated

No user intervention necessary once we start running things
Targeted Automated Testing Using Constraint Logic Programming

Targeted

Not completely random; trying to create specific inputs which act as good tests
Targeted Automated Testing Using Constraint Logic Programming

99% Prolog, plus some other nice features
Outline

• Background
• Research problem
• Applications
  • Data Structure Generation
  • Generating JavaScript Programs with Known Runtime Behaviors
  • Testing Rust’s Typechecker
  • Testing SMT Solvers
• Conclusion
Outline

• Background

• Research problem

• Applications
  • Data Structure Generation
  • Generating JavaScript Programs with Known Runtime Behaviors
  • Testing Rust’s Typechecker
  • Testing SMT Solvers

• Conclusion
Automated Testing

Motivation

• Writing correct software is hard
• Writing tests is time-consuming
• CPU cycles are cheap
Background: Differential Testing

- Idea: generate an input via some process
- Run input on different implementations
- If implementations disagree on result, bug has been found
Input Generator
function foo() { ... }

... bar();
function foo() {
    ...
}

...
function foo() {
    ... 
}

bar();

Test Input
Executed on

42
42
53

Produce
4
function foo() {
  ... 
}...
bar();

Test Input
Executed on

42

Mismatch: bug 53
function foo() { ... } ... bar();

My research

Test Input

Executed on

42

42

53

Produce
Outline

• Background

• **Research problem**

• Applications
  • Data Structure Generation
  • Generating JavaScript Programs with Known Runtime Behaviors
  • Testing Rust’s Typechecker
  • Testing SMT Solvers

• Conclusion
Consider the JavaScript snippet from before:

```javascript
function foo() { ... }
...
bar();
```
Any random input for “...” forms a possible test

```javascript
function foo() {
    qw
}
asdf
bar();
```
Any random input for “...” forms a possible test

```javascript
function foo() { qw }
asdf
bar();
```

...but there is no telling if this will be a good test
function foo() {
  qw
} 

...but there is no telling if this will be a good test 

(In this case, at best a test of variable lookup)
More complex example: a library that manipulates red-black trees
More complex example: a library that manipulates red-black trees
More complex example: a library that manipulates red-black trees

Intuitively, we need at least a significant portion of tests which consist of valid red-black trees
More complex example: a library that manipulates red-black trees

Intuitively, we need at least a significant portion of tests which consist of valid red-black trees

Hard problem!
More complex example: a library that manipulates red-black trees

Intuitively, we need at least a significant portion of tests which consist of valid red-black trees

Still too trivial!
Too trivial?

- Arbitrary red-black trees aren’t bad tests, but they aren’t good tests, either

- More interesting: valid red-black trees of depth $\leq D$, containing values between 0 and $K$, which will rebalance on the insertion of value $V$
Too trivial?

- Arbitrary red-black trees aren’t bad tests, but they aren’t good tests, either.

- More interesting: valid red-black trees of depth $\leq D$, containing values between 0 and $K$, **which will rebalance on the insertion of value** $V$.

  - Tests are much more specific.
  - Targeted tests for finding bugs in insertion and rebalancing.
  - Significantly more difficult.
  - Unique to my work.
Key Insights (1)

• Valid test inputs can be described as solutions to systems of logical constraints
Key Insights (1)

- Valid test inputs can be described as solutions to systems of logical constraints
Key Insights (1)

• Valid test inputs can be described as solutions to systems of logical constraints
Key Insights (2)

• A variety of search strategies can be used to explore the space of solutions to these logical constraints
Key Insights (2)

• A variety of search strategies can be used to explore the space of solutions to these logical constraints

  • Search strategies can be independent of constraints

  • Search strategies are useful because there tend to be many, even infinitely many, solutions

• E.g., depth-first search, random, etc.
Enter Constraint Logic Programming

• Constraint Logic Programming (CLP) is Prolog integrated with arithmetic constraint solvers

• CLP overall is viewable as a solver of logical constraints, with fine-grained control over how the constraints are solved

• Therefore, we can specify test input constraints in CLP, and use existing CLP engines to generate corresponding inputs
Digression: Why CLP?

- Why **not** SMT solvers?
  - $x > y \land y < z$
  - Not designed for getting *all* solutions, only *one* solution; getting all ranges from practically to actually impossible
  - Slow (testing faster than generation)
  - No / minimal control over search
Digression: Why CLP?

• Why not a custom constraint solver?
  • Lots of engineering needed to make it fast, which is unrelated to the test generation problem
  • Very easy to accidentally reimplement CLP
Digression: Why CLP?

- Why not a custom constraint solver?
  - Lots of engineering needed to make it fast, which is unrelated to the test generation problem
  - Very easy to accidentally reimplement CLP


CLP Example: Binary Trees

- Binary trees, **not** binary search trees
- Consist of leaves and internal nodes
  - Both are associated with one value
- Constraint: for each element value \( E \),
  \[ \text{Min} \leq E \leq \text{Max} \]
CLP Example: Binary Trees

- Binary trees, not binary search trees
- Consist of leaves and internal nodes
  - Both are associated with one value
- Constraint: for each element value $E$, $\text{Min} \leq E \leq \text{Max}$
CLP Example: Binary Trees

• Binary trees, **not** binary search trees

• Consist of leaves and internal nodes
  • Both are associated with one value

• **Constraint:** for each element value $E$, $\text{Min} \leq E \leq \text{Max}$

```prolog
inBounds(E, Min, Max) :-
    Min #=< E,
    E #=< Max.
```
CLP Example: Binary Trees

- Binary trees, **not** binary search trees
- Consist of leaves and internal nodes
  - Both are associated with one value
- Constraint: for each element value $E$, $\text{Min} \leq E \leq \text{Max}$

Clause - comparable to function definition

```prolog
inBounds(E, Min, Max) :-
    Min #=< E,
    E #=< Max.
```
CLP Example: Binary Trees

• Binary trees, **not** binary search trees

• Consist of leaves and internal nodes
  
  • Both are associated with one value

• **Constraint:** for each element value $E$, $\text{Min} \leq E \leq \text{Max}$

Clause head - comparable to function signature

```
inBounds(E, Min, Max) :-
    Min #=< E,
    E #=< Max.
```
CLP Example: Binary Trees

- Binary trees, **not** binary search trees
- Consist of leaves and internal nodes
  - Both are associated with one value
- **Constraint:** for each element value \( E \), \( \text{Min} \leq E \leq \text{Max} \)

```
inBounds(E, Min, Max) :-
    Min #=< E,
    E #=< Max.
```

Clause body - comparable to function body
CLP Example: Binary Trees

- Binary trees, **not** binary search trees
- Consist of leaves and internal nodes
  - Both are associated with one value
- **Constraint:** for each element value $E$, $\text{Min} \leq E \leq \text{Max}$

\[
\text{inBounds}(E, \text{Min}, \text{Max}) \overset{\text{:-}}{=} \text{Min} \#=< E, \quad \text{Reverse implication}
\text{E} \#=< \text{Max}. \quad (\Leftarrow)
\]
CLP Example: Binary Trees

- Binary trees, not binary search trees
- Consist of leaves and internal nodes
  - Both are associated with one value
- Constraint: for each element value \( E \),
  \[
  \text{Min} \leq E \leq \text{Max}
  \]

\[
\text{inBounds}(E, \text{Min}, \text{Max}) :- \\
\text{Min} \#=< E, \\
E \#=< \text{Max}.
\]

Conjunction \((\land)\)
CLP Example: Binary Trees

• Binary trees, not binary search trees
• Consist of leaves and internal nodes
  • Both are associated with one value
• Constraint: for each element value $E$, $\text{Min} \leq E \leq \text{Max}$

\[\text{inBounds}(E, \text{Min}, \text{Max}) :-
\begin{align*}
\text{Min} & \#=< E, \\
E & \#=< \text{Max}.
\end{align*}\]

Arithmetic $\leq$ over symbolic variables
CLP Example: Binary Trees

- Binary trees, **not** binary search trees
- Consist of leaves and internal nodes
  - Both are associated with one value
- **Constraint:** for each element value E,
  \[ \text{Min} \leq E \leq \text{Max} \]

\[
\text{inBounds}(E, \text{Min}, \text{Max}) :- \\
\text{Min} \#=< E, \\
E \#=< \text{Max}.
\]

End of clause
CLP Example: Binary Trees

- Binary trees, **not** binary search trees
- Consist of leaves and internal nodes
  - Both are associated with one value
- **Constraint:** for each element value \( E \),
  \[
  \text{Min} \leq E \leq \text{Max}
  \]

\[
\text{inBounds}(\text{Elem}, \text{Min}, \text{Max}) \ :-
\text{Min} \#=< \text{Elem},
\text{Elem} \#=< \text{Max}.
\]

**Logical meaning:**
\[
\forall \text{Min} . \forall \text{Elem} . \forall \text{Max} .
\text{inBounds}(\text{Elem}, \text{Min}, \text{Max}) \iff 
\text{Min} \leq \text{Elem} \land \text{Elem} \leq \text{Max}
\]
CLP Example: Binary Trees

• Binary trees, not binary search trees

• Consist of leaves and internal nodes
  • Both are associated with one value

• Constraint: for each element value $E$, $\text{Min} \leq E \leq \text{Max}$
inBounds(Elem, Min, Max) :-
    Min #=< Elem,
    Elem #=< Max.
CLP Example: Binary Trees

\[
in\text{Bounds}(\text{Elem}, \text{Min}, \text{Max}) :\neg
\]
\[
\quad \text{Min} \#=< \text{Elem},
\]
\[
\quad \text{Elem} \#=< \text{Max}.
\]

\[
tree(leaf(\text{Elem}), \text{Min}, \text{Max}) :\neg
\]
\[
\quad \text{in}\text{Bounds}(\text{Elem}, \text{Min}, \text{Max}).
\]
inBounds(Elem, Min, Max) :-
    Min #=< Elem,
    Elem #=< Max.

tree(leaf(Elem), Min, Max) :-
    inBounds(Elem, Min, Max).
inBounds(Elem, Min, Max) :-
    Min #=< Elem,
    Elem #=< Max.

tree(leaf(Elem), Min, Max) :-
inBounds(Elem, Min, Max).
CLP Example: Binary Trees

\[
inBounds(Elem, Min, Max) :-
    Min #=< Elem,
    Elem #=< Max.
\]

\[
tree(leaf(Elem), Min, Max) :-
    inBounds(Elem, Min, Max).
\]

\[
tree(node(Left, Elem, Right), Min, Max) :-
    inBounds(Elem, Min, Max),
    tree(Left, Min, Max),
    tree(Right, Min, Max).
\]
inBounds(Elem, Min, Max) :-
    Min #=< Elem,  
    Elem #=< Max.

2

1

tree(leaf(Elem), Min, Max) :-
    inBounds(Elem, Min, Max).

2

1

tree(node(Left, Elem, Right), 
    Min, Max) :-
    inBounds(Elem, Min, Max), 
    tree(Left, Min, Max), 
    tree(Right, Min, Max).

1
CLP Example: Binary Trees

\[
inBounds(Elem, Min, Max) :-
    Min #=< Elem,
    Elem #=< Max.
\]

\[
tree(leaf(Elem), Min, Max) :-
    inBounds(Elem, Min, Max).
\]

\[
tree(node(Left, Elem, Right), Min, Max) :-
    inBounds(Elem, Min, Max),
    tree(Left, Min, Max),
    tree(Right, Min, Max).
\]
inBounds(Elem, Min, Max) :-
    Min #=< Elem,
    Elem #=< Max.

tree(leaf(Elem), Min, Max) :-
    inBounds(Elem, Min, Max).

tree(node(Left, Elem, Right),
    Min, Max) :-
    inBounds(Elem, Min, Max),
    tree(Left, Min, Max),
    tree(Right, Min, Max).
CLP Example: Binary Trees

\[
\text{inBounds(Elem, Min, Max) :-}
\text{Min #=< Elem,}
\text{Elem #=< Max.}
\]

\[
\text{tree(leaf(Elem), Min, Max) :-}
\text{inBounds(Elem, Min, Max).}
\]

\[
\text{tree(node(Left, Elem, Right), Min, Max) :-}
\text{inBounds(Elem, Min, Max),}
\text{tree(Left, Min, Max),}
\text{tree(Right, Min, Max).}
\]
CLP Example: Binary Trees

• Generating valid trees can be done like so:

```prolog
?- tree(Tree, 0, 3), writeln(Tree), fail.
```
CLP Example: Binary Trees

- Generating valid trees can be done like so:

?- tree(Tree, 0, 3), writeln(Tree), fail.
CLP Example: Binary Trees

• Generating valid trees can be done like so:

```
?- tree(Tree, 0, 3),
   writeln(Tree),
   fail.
```

Generate a tree with Min = 0 and Max = 3; bind it to Tree
CLP Example: Binary Trees

• Generating valid trees can be done like so:

```prolog
?- tree(Tree, 0, 3), writeln(Tree), fail.
```

Write out the tree
CLP Example: Binary Trees

• Generating valid trees can be done like so:

```
?- tree(Tree, 0, 3),
    writeln(Tree),
    fail.
```

Trigger backtracking to occur to generate another tree.

Intuitively, Tree is nondeterministically bound to all possible trees.
Outline

• Background

• Research problem

• Applications
  • Data Structure Generation
  • Generating JavaScript Programs with Known Runtime Behaviors
  • Testing Rust’s Typechecker
  • Testing SMT Solvers

• Conclusion
Data Structure (DS) Generation

• Applied CLP to the generation of complex data structures, along with particular variants of interest for testing
  • Variants form a strict subset of the space, and each DS had its own variant
• Most of the data structures were novel to our work, along with all of the variants
  • Intentionally wanted to push the limit
Data Structures

• Sorted linked lists
• Red-black trees
• Array-based heaps (priority queues)
• ANI images (via grammars)
• Skip lists
• Splay trees
• B-trees
Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees
Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees
Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees
Data Structures

- Sorted linked lists
- Red-black trees  
  Covered in related work
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees
Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Novel to our work
Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees
Data Structures

- Sorted linked lists
- Red-black trees
- Array-based heaps (priority queues)
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees
Data Structures

• Sorted linked lists
• Red-black trees
• Array-based heaps (priority queues)
• ANI images (via grammars)
• Skip lists
  • Splay trees
• B-trees
Data Structures

• Sorted linked lists
• Red-black trees
• Array-based heaps (priority queues)
• ANI images (via grammars)
• Skip lists
• Splay trees
• B-trees
Example Variant (Same as Previously Described)

• Valid red-black trees of depth $\leq D$, containing values between 0 and $K$, which will rebalance on the insertion of value $V$. 
Example Variant (Same as Previously Described)

- Valid red-black trees of depth \( \leq D \), containing values between 0 and \( K \), which **will rebalance on the insertion of value** \( V \)

\[
isBST(\text{Tree}1) \land 
isRedBlackTree(\text{Tree}1) \land 
callsRebalance( 
    insert(\text{Tree}1, V, \text{Tree}2))
\]
Example Variant (Same as Previously Described)

- Valid red-black trees of depth $\leq D$, containing values between 0 and $K$, which will rebalance on the insertion of value $V$

\[
\text{isBST(Treel)} \land \\
\text{isRedBlackTree(Treel)} \land \\
\text{callsRebalance(}
\text{insert(Treel, } V, \text{ Tree2}))
\]
Example Variant (Same as Previously Described)

- Valid red-black trees of depth $\leq D$, containing values between 0 and $K$, which will rebalance on the insertion of value $V$

\[
\text{isBST}(\text{Tree1}) \land \\
\text{isRedBlackTree}(\text{Tree1}) \land \\
\text{callsRebalance}(\text{insert}(\text{Tree1}, V, \text{Tree2}))
\]
Example Variant (Same as Previously Described)

• Valid red-black trees of depth \( \leq D \), containing values between 0 and \( K \), which will rebalance on the insertion of value \( V \)

\[
isBST(Tree1) \land \\
isRedBlackTree(Tree1) \land \\
callsRebalance(insert(Tree1, V, Tree2))
\]
Example Variant (Same as Previously Described)

• Valid red-black trees of depth \( \leq D \), containing values between 0 and \( K \), which will rebalance on the insertion of value \( V \)

\[
isBST(\text{Tree1}) \land isRedBlackTree(\text{Tree1}) \land \text{callsRebalance}(\text{insert}(\text{Tree1}, V, \text{Tree2}))
\]

Requires reasoning about both rebalancing and insertion; intuitively large state space.
Example Variant (Same as Previously Described)

- Valid red-black trees of depth \( \leq D \), containing values between 0 and \( K \), which will rebalance on the insertion of value \( v \)

Naive approach: generate all possible trees, and filter those that are related via insert

\[
\text{isBST(Tree1)} \land \text{isRedBlackTree(Tree1)} \land \text{callsRebalance(insert(Tree1, V, Tree2))}
\]
Evaluation

• Tested all aforementioned data structures and their special variants on Korat, UDITA, and CLP (using GNU Prolog)

• Measured **how quickly** all data structures within certain bounds (small, medium, large) could be generated, with a 30 minute timeout

  • Quicker generation means more time testing and less time generating
Small Bounds

Seconds (lower is better)

UDITA Is Extremely Slow
Seconds (lower is better)

Small Bounds

- Korat
- CLP

- Lists
- Red-Black
- Heaps
- Image
- Skip
- Splay
- B-Trees

Bar chart showing performance of various data structures in seconds, with lower values indicating better performance.
Small Bounds

Seconds (lower is better)

CLP Barely Registers
Medium Bounds

- UDITA times out on everything
- Korat times out on 5 / 14 experiments
- CLP is generally $\sim 30 \times - 1,000 \times$ faster
- For B-trees, Korat and UDITA both timeout, but CLP completes within **a single millisecond**, ultimately thanks to the **capability to control search**
  - Internally, they took the naive strategy
Large Bounds

- Korat and UDITA timeout on everything
- Depending on the data structure, CLP takes between ~70 seconds and just under 30 minutes
Outline

- Background
- Research problem
- Applications
  - Data Structure Generation
  - Generating JavaScript Programs with Known Runtime Behaviors
    - Testing Rust’s Typechecker
    - Testing SMT Solvers
- Conclusion
Generating Programs

• Bulk of existing literature is focused on stochastic grammars
  • Randomly walk over a language’s grammar, producing syntactically valid programs as a result
Generating Programs

- Bulk of existing literature is focused on stochastic grammars
  - Randomly walk over a language’s grammar, producing syntactically valid programs as a result

\[ e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2 \]
Generating Programs

• Bulk of existing literature is focused on *stochastic grammars*
  
  • Randomly walk over a language’s grammar, producing syntactically valid programs as a result

\[ e \in ArithExp ::= n \in \mathbb{N} \mid e_1 + e_2 \]
Generating Programs

• Bulk of existing literature is focused on stochastic grammars
  • Randomly walk over a language’s grammar, producing syntactically valid programs as a result

\[ e \in ArithExp ::= n \in \mathbb{N} \mid e_1 + e_2 \]
Generating Programs

• Bulk of existing literature is focused on stochastic grammars
  
  • Randomly walk over a language’s grammar, producing syntactically valid programs as a result

\[ e \in ArithExp ::= n \in \mathbb{N} \mid e_1 + e_2 \]
Generating Programs

• Bulk of existing literature is focused on \textit{stochastic grammars}

  • Randomly walk over a language’s grammar, producing syntactically valid programs as a result

\[
e \in \text{ArithExp} ::= n \in \mathbb{N} \quad | \quad e_1 + e_2
\]
Example Derivation

\[ e \in ArithExp ::= n \in \mathbb{N} \quad | \quad e_1 + e_2 \]

\[ e_3 + e_4 \quad 3 \]

\[ 1 \quad 3 \]
Example Derivation

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

![Diagram showing the derivation tree with nodes labeled as follows:
- e
- e_1 + e_2
- e_3 + e_4
- 3
- 1
- 3

The nodes are connected by arrows indicating the derivation process.](image-url)
Problems with Stochastic Grammars

• All you get is syntactic validity
  • No idea what programs do
  • Programs are not generally well-typed
  • Difficult to test particular components (e.g., specifically code generation)
  • Only configuration is by tuning probabilities
Enter CLP

- Generating syntactically valid programs is easy...
Syntactic Validity

\[ e \in ArithExp ::= n \in \mathbb{N} \mid e_1 + e_2 \]
Syntactic Validity

\[ e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2 \]
Syntactic Validity

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

\[
\text{arithExp}(\text{num}(N)) :- \\
\text{INTMIN} \leq N, \\
N \leq \text{INTMAX}.
\]
Syntactic Validity

\[ e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2 \]

\[
\text{arithExp}(\text{num}(N)) \ :- \\
\text{INTMIN} \text{ } \#=< \text{ } N, \\
N \text{ } \#=< \text{ } \text{INTMAX}.
\]
Syntactic Validity

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

\[
\text{arithExp}(\text{num}(N)) : - \\
\text{INTMIN} \leq N, \\
N \leq \text{INTMAX}.
\]
Syntactic Validity

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

\[
\text{arithExp}(\text{num}(N)) :- \\
\text{INTMIN} \#=< N, \\
N \#=< \text{INTMAX}.
\]
Syntactic Validity

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

\[
\begin{align*}
\text{arithExp(num(N))} & : - \\
& \text{INTMIN } \#=< N, \\
& \text{N } \#=< \text{INTMAX}.
\end{align*}
\]
Syntactic Validity

\[ e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2 \]

\[
\text{arithExp}(\text{num}(N)) :- \\
\quad \text{INTMIN} \leq N, \\
\quad N \leq \text{INTMAX}.
\]

\[
\text{arithExp}(\text{add}(E1, E2)) :- \\
\quad \text{arithExp}(E1), \\
\quad \text{arithExp}(E2).
\]
Syntactic Validity

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

\[
\text{arithExp}(\text{num}(N)) :- \\
\quad \text{INTMIN} \lea N, \\
\quad \text{N} \lea \text{INTMAX}.
\]

\[
\text{arithExp}(\text{add}(E1, E2)) :- \\
\quad \text{arithExp}(E1), \\
\quad \text{arithExp}(E2).
\]
Syntactic Validity

\[ e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2 \]

\[
\text{arithExp}(\text{num}(N)) :-
\]
\[
\text{INTMIN} \leq N,
\]
\[
N \leq \text{INTMAX}.
\]

\[
\text{arithExp}(\text{add}(E1, E2)) :-
\]
\[
\text{arithExp}(E1),
\]
\[
\text{arithExp}(E2).
\]
Syntactic Validity

\[ e \in \text{ArithExp} ::= n \in \mathbb{N} \mid e_1 + e_2 \]

\[
\text{arithExp}(\text{num}(N)) ::= \\
\begin{align*}
\text{INTMIN} & \leq N, \\
N & \leq \text{INTMAX}.
\end{align*}
\]

\[
\text{arithExp}(\text{add}(E_1, E_2)) ::= \\
\text{arithExp}(E_1), \\
\text{arithExp}(E_2).
\]
Beyond Syntax

- For example, programs which evaluate to a particular value
  - A semantic property
- Involves writing an evaluator for the language in CLP
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).
Expressions that Evaluate to 7

```prolog
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).
```
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).
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).
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).
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).
Expressions that Evaluate to 7

\[
\text{eval}(\text{num}(N), N).
\]
\[
\text{eval}(\text{add}(E_1, E_2), N) :-
\]
\[
\text{eval}(E_1, N_1),
\]
\[
\text{eval}(E_2, N_2),
\]
\[
N \neq N_1 + N_2.
\]

% same arithExp from before
\[\text{evalsTo7}(E) :-\]
\[\text{arithExp}(E),\]
\[\text{eval}(E, 7).\]
Expressions that Evaluate to 7

\[
\begin{align*}
\text{eval} & (\text{num}(N), N). \\
\text{eval}(\text{add}(E1, E2), N) & : - \\
& \quad \text{eval}(E1, N1), \\
& \quad \text{eval}(E2, N2), \\
& \quad N \# = N1 + N2. \\
\end{align*}
\]

% same arithExp from before
\[
\text{evalsTo7}(E) : - \\
\text{arithExp}(E), \\
\text{eval}(E, 7).
\]
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).
Scaling Up

• This central idea was applied to generating JavaScript programs with known runtime behaviors

• Compared the generation rate to that of a finely-tuned stochastic grammar designed for the same thing

  • Stochastic grammar probabilities were tuned to try to generate programs with certain behaviors
Scaling Up

• For JavaScript, generate programs which:
  • avoid dereferencing `null`: CLP $\sim 3.8x$ faster
  • stress integer optimizations: CLP $\sim 7.8x$ faster
  • utilize `with` and higher-order functions in problematic ways: CLP $\sim 3.1 \text{ million } x$ faster
  • utilize prototype-based inheritance: CLP `infinitely` faster (stochastic grammars never generated such a program within the five-minute timeframe)
Outline

- Background
- Research problem

- Applications
  - Data Structure Generation
  - Generating JavaScript Programs with Known Runtime Behaviors
  - Testing Rust’s Typechecker
  - Testing SMT Solvers
- Conclusion
Why Rust?

• A real language with a rapidly growing user base (over 6,600 packages available currently)

• A sophisticated type system with important guarantees (e.g., memory safety without GC)

• No formal semantics, or even an informal specification
  • Worked closely with Rust development team
Why Rust’s Typechecker?

- Well-typed Rust programs are memory safe
- If the typechecker fails to flag an ill-typed program, then there is a *silent loss* of memory safety guarantees
- Most complex language component at the time
Unique Challenge: There is Only One Rust
function foo() {
    ...
}

... bar();

Test Input
Executed on

Firefox: 42
Google Chrome: 42
Internet Explorer: 53

Produce: 33
Input Generator

Generates

fn foo() { ... }
fn bar() { ... }
fn main() { ... }

Test Input

Executed on well-typed

well-typed
Input Generator

Generates

Test Input

Executed on

well-typed

Nothing to compare to!

fn foo() { ... }
fn bar() { ... }
fn main() { ... }
Solution

• Generate tests which behave predictably ahead of time, and check that the underlying system agrees with the predication

• In other words, generate tests which we know to be well-typed or ill-typed, and make sure Rust agrees

• Requires understanding Rust’s type system
Rust’s Type System

• Rust has typeclasses, parametric polymorphism, generics, and affine types for guaranteeing memory safety statically

• Handling affine types properly requires:
  • Symbolic arithmetic constraints
  • Constraints on type variables
Rust’s Type System

• Rust has typeclasses, parametric polymorphism, generics, and affine types for guaranteeing memory safety statically.

• Handling affine types properly requires:
  • Symbolic arithmetic constraints
  • Constraints on type variables
Rust’s Type System

• Rust has typeclasses, parametric polymorphism, generics, and **affine types** for guaranteeing memory safety statically

• Handling affine types properly requires:
  • **Symbolic arithmetic constraints**
  • **Constraints on type variables** Unavailable in existing systems
Rust’s Type System

- Rust has typeclasses, parametric polymorphism, generics, and **affine types** for guaranteeing memory safety statically

- Handling affine types properly requires:
  - Symbolic arithmetic constraints
  - **Constraints on type variables**

*Never attempted before*
Rust’s Type System

- Rust has typeclasses, parametric polymorphism, generics, and **affine types** for guaranteeing memory safety statically.

- Handling affine types properly requires:
  - Symbolic arithmetic constraints
  - Constraints on type variables

Altogether, must embed a specialized constraint solver for handling these features in CLP itself.
Well-Typed for Testing

• For testing purposes, well-typed programs are not particularly interesting
  • If compiler rejects a well-typed program, the programmer gets annoyed
  • Types as analysis: rejection of a well-typed program is a precision issue
Ill-Typed for Testing

• More interesting for testing purposes: ill-typed programs
  • If a compiler accepts an ill-typed program, we get a silent loss of guarantees
  • For Rust, this means programs are not necessarily memory-safe, defeating the entire purpose of the language
• Types as analysis: accepting an ill-typed program is a soundness issue
Generating Ill-Typed Programs

- Naive approach: generate syntactically valid programs and discard those that happen to be well-typed
  - Relatively efficient (most will be ill-typed)
  - Most programs are *obviously* ill-typed (multiple type errors; a typechecker need only spot one, so this masks bugs)
Generating Ill-Typed Programs

- Better approach: generate programs which are *almost* well-typed

- Intuitively, negate a single premise in a typing rule, leading to programs which are ill-typed by construction, but only with respect to the single negated premise

  - Results in highly targeted tests

- This idea is novel, and this was the first attempt to generate anything intentionally ill-typed
Rust Testing Results

• Able to generate ~2,300 programs per second

  • Versus ~2 per second compared to preexisting techniques on a simpler language CLP is over 1,100x faster for a significantly more complex language

• Found 18 issues; developers considered 14 of these bugs

  • Included one specification level bug
Outline

• Background

• Research problem

• Applications
  • Data Structure Generation
  • Generating JavaScript Programs with Known Runtime Behaviors
  • Testing Rust’s Typechecker
  • Testing SMT Solvers

• Conclusion
SMT Solvers

• Used for solving constraints specified in the SMT-LIB language, in a similar vein as CLP

• Crucially important in software verification; that is, proving some code does the right thing

• Solvers can be buggy too

• Buggy solvers can mean faulty proofs
Testing Approach

• Lots of details
• Work is currently in submission
• Basic idea: test with well-typed SMT-LIB formulas which are known to be logically satisfiable or unsatisfiable ahead of time
Testing Results

- 24 bugs have been found across a number of solvers

- **Every solver tested had at least one correctness bug, including Z3**

- Included a *specification bug* which required communication with the standards committee
Outline

- Background
- Research problem
- Applications
  - Data Structure Generation
  - Generating JavaScript Programs with Known Runtime Behaviors
  - Testing Rust’s Typechecker
  - Testing SMT Solvers
- Conclusion
Conclusion

- CLP is applicable to a number of test problems
- CLP is capable of generating very complex tests in a high-performance fashion
- No need to write lots of test generation code with CLP