Breccia: A Functional DSL Compiled to Egglog for Test Input Generation

Peter Boyland[1], Sarah Canto Hyatt[1], Kyle Dewey[2], and Ben Hardekopf[1]

[1] University of California Santa Barbara, Santa Barbara CA, 93106, USA
[2] California State University Northridge, Northridge CA, 91330, USA

Abstract. Some prominent efforts within fuzz testing to automatically generate test inputs with complex invariants (e.g., well-typed programs for fuzzing compilers) have exploited logical satisfaction, such as using Prolog to describe constraints and generate satisfying instances. However, using Prolog has met with resistance from the community due to the difficulty of writing Prolog descriptions with acceptable performance. We propose an alternative logic programming-based approach: Datalog. While Datalog improves on the "effective programmability" of Prolog, it has its own issues: it is less expressive than Prolog, its bottom-up evaluation strategy requires keeping all generated terms in memory, and while it is easier to program effective generators in Datalog than Prolog, it still requires a specific style and has hidden pitfalls (such as bounding). We introduce Breccia, a functional DSL for generating objects with complex invariants that compiles to Egglog, a Datalog language extended with equality saturation. We show that Breccia is expressive enough to describe interesting objects such as well-typed programs with non-trivial type systems, and that equality saturation can greatly improve the performance of the resulting Egglog generator. We compare Breccia against Prolog implementations of similar generators in order to understand the tradeoffs involved.

Keywords: Generation · Prolog · Datalog · Equality Saturation · Fuzzing

1 Introduction

The problem that we address in this paper is test input generation for fuzzing systems under test. While some systems can be profitably fuzzed using random bitstrings as inputs [27], others require inputs satisfying complex constraints. For example, if fuzzing a compiler, it is unlikely that a randomly-generated bitstring is a syntactically valid and well-typed program, and such programs are needed to test deeper compiler components. One way to generate inputs satisfying constraints is to treat generation as a logic satisfiability problem, encoding input constraints as SAT or SMT formulas [26, 36] or Prolog programs [11, 13–15].

Our attention is on the last strategy: writing Prolog programs to check if constraints on an input hold, and then running the program "backwards" to generate inputs satisfying those constraints. This is a theoretically elegant solution; for

example, we can write a typechecker in Prolog and use it both to typecheck existing programs or create new ones. However, the reality is unfortunately not so elegant. Prolog's execution model is very sensitive to how programs are written, including constraint order and how constraints are described. Directly transcribing systems of constraints (like type systems) into Prolog is unlikely to create a usable generator; the program is more likely to run infinitely without generating any satisfying terms. In practice, writing Prolog-based generators requires in-depth reasoning about the input domain being generated, and Prolog programs must be carefully designed and optimized in ways that make generators look and behave very differently from acceptors. This issue has given Prolog a reputation for being difficult to use for test input generation [3, 6, 19].

In this paper we investigate a different approach to using logic programming for test input generation: using Datalog rather than Prolog. Datalog's bottom-up evaluation strategy (as opposed to Prolog's top-down strategy) means that it is much less sensitive to how a program is written. Datalog also features a fixpoint semantics wherein all satisfying terms and subterms are kept in memory, avoiding recomputation if any subterm is needed again. Even simple fuzzing problems reuse subterms, and memoization dramatically improves fuzzer performance [12, 22]. With Datalog, this memoization is transparent and fully automatic. Of course, Datalog is also less expressive than Prolog, leading to our first research question: **RQ1: is Datalog expressive enough to describe interesting, non-trivial constraints for test input generation?**

Datalog's fixpoint semantics is a double-edged sword, because it requires all terms to be kept in memory. This contrasts with Prolog, whose depth-first search strategy means that as satisfying terms are computed, they can be output and then "forgotten", without consuming further memory. Our answer to this problem is to use *e-graphs* [29], as used in *equality saturation* [38], to store computed terms. E-graphs are a data structure for compactly storing congruence classes, and we exploit their capabilities to address Datalog's higher memory usage. In fact, e-graphs can do more than just reduce memory consumption: by defining congruence classes specific to the domain being generated, we should be able to significantly improve the generation rate. For example, if generating well-typed programs, we could define types as congruence classes so that all `int`-typed expressions are considered equivalent. If some expression e_1 uses an `int`-typed subexpression e_2, then our generator only needs to generate e_1 using the `int` congruence class rather than all expressions in that congruence class. This approach leads to our second research question: **RQ2: Can e-graphs and equality saturation significantly improve the performance of a Datalog-based generator?**

While Datalog is easier to use than Prolog for describing generators, Datalog still requires a rather particular style to transcribe constraints into programs. Datalog has pitfalls that can be easy to fall into, such as properly bounding generation to ensure termination. To address this issue we follow other work that has created domain-specific functional languages that compile to Datalog [30]. In our case, we have designed *Breccia*: a functional DSL for writing constraint-

based generators. Breccia compiles to Egglog [42], a Datalog implementation that incorporates e-graphs and equality saturation. The compilation to Egglog automatically ensures bounding and inserts proper uses of equality saturation, based on the Breccia description.

Both the Prolog and Breccia generators that we investigate are *bounded exhaustive* generators, that is, they confine themselves to a finite subset of the infinite space of possible terms and fully explore that finite space. Bounded exhaustive generation relies on the *small scope hypothesis* [21], which states that a high percentage of bugs can be revealed by exploring all test inputs within some small scope (e.g., bounded by size of input). To compare the Prolog and Breccia generators, we investigate their respective generation rates and also run a mini-fuzzing campaign on three of our example languages to demonstrate their ability to detect bugs.

In the remainder of this paper we:

- Describe how to use Datalog for test input generation (Section 2);
- Describe how to optimize Datalog generation using e-graphs (Section 3);
- Describe the Breccia generator DSL and compiler (Section 4);
- Evaluate Breccia by comparing it against Prolog-based generators for a series of statically-typed languages including simply-typed lambda calculus, System F, and Featherweight Java (Section 5);
- Discuss various related works (Section 7).

The Breccia implementation and all generators used in the evaluation are available at `https://github.com/pllab/breccia`.

2 Generation Using Datalog

Datalog[3] generation is a good fit for constraints that are *compositional*, i.e., a term's constraint-satisfaction depends only on its subterms and how it composes them. For example, type systems are defined using inductive rules over the AST, such that a term's well-typedness depends on the types of its subterms and how they are composed into a new term. Other possible good fits include sorted lists, binary search trees, red-black trees, heaps, and B-trees [13, 22, 35]. On the other hand, non-compositional problems, such as generating specific integers or bit-strings matching some user-specified criteria, would be a poor fit, as would any problem with an unpredictable global context. We will illustrate Datalog generation using the running example of the simply-typed lambda calculus (STLC) [7]; see Figure 1a for its abstract syntax and Figure 1b for its typing rules. We incrementally build a Datalog generator for well-typed STLC terms in the rest of this section.[4] We demonstrate that this process is straightforward, though it requires some extra bookkeeping. While STLC is simple, our evaluation (Section 5) shows that more complex type systems can be handled similarly.

[3] We assume a modern version of Datalog extending the original definition [5] with features like algebraic datatypes (e.g., Egglog [42]). This added expressiveness comes at the cost of possible non-termination; the programmer must ensure termination.

[4] The description is abstracted for space; see Appendix A for the full code listing.

$$\tau \in \mathit{Type} ::= \iota \mid \tau \to \tau$$
$$e \in \mathit{Exp} ::= x \in \mathit{Var} \mid \lambda\, x{:}\tau\, .\, e \mid e_1\ e_2$$

(a) Syntax (where ι is some base type).

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \textsc{var}$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda\, x{:}\tau_1\, .\, e : \tau_1 \to \tau_2}\ \textsc{abs}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{app}$$

(b) Typing (where $\Gamma : \mathit{Var} \to \mathit{Type}$).

Fig. 1: Simply-Typed λ-Calculus

2.1 Generation Strategy

We begin by defining datatypes for the terms being generated; for STLC this is `Type` (Figure 2) and `Exp` (Figure 3). We deviate from the formal definition by associating each variable with its type explicitly, making generation easier as (1) we do not need to track the context Γ; and (2) since we generate terms bottom-up, variables are effectively used before being declared, so they must have known types. Variable names are represented with integers instead of strings.

```
datatype Type =
| Base
| Arr(Type, Type)
```

Fig. 2: STLC Types

```
datatype Exp =
| Var(Int, Type)
| Abs(Var, Exp)
| App(Exp, Exp)
```

Fig. 3: STLC Expressions

The general strategy for a Datalog generator is to start with the base cases of an inductive definition and then build larger, more complex terms while ensuring that each new term satisfies the required contraints. Each base case is an initial fact, and each inductive case is a rule taking existing terms and building bigger terms. We start with types and defining the predicate `type` : `Type`, i.e., `type` will be a Datalog relation stating that its argument is a well-defined STLC type per Figure 2. According to Figure 2, `Base` is a well-formed type, giving us our initial fact `type(Base)`. Also from Figure 2, if `type(t1)` and `type(t2)`, then `type(Arr(t1,t2))`, giving us a rule composing types into larger types.

For types we only need well-formedness, but for terms we need both well-formedness and well-typedness. The typing judgement defined in Figure 1b is expressed with a predicate `well-typed` : `Exp` × `Type`; note that since we explicitly associate each variable with its type we do not need the Γ argument, so the Datalog relation `well-typed` only takes an STLC expression and its type. The base case is VAR, so the rule is that if `i` is an integer and `type(t)` then `well-typed(Var(i,t), t)`; that is, the type of some variable `i` associated with

type `t` is `t`. Thus each generated type has some number of variables associated with it.

Finally, we translate the ABS and APP rules to build larger well-typed terms. For ABS, the rule is that if `well-typed(Var(i,t1),t1)` and `well-typed(e, t2)`, then `well-typed(Abs(Var(i,t1),e), Arr(t1,t2))`. For APP, the rule is that if `well-typed(e1,Arr(t1,t2))` and `well-typed(e2,t2)`, then `well-typed(App(e1,e2),t2)`. With these rules, we can generate an unbounded number of well-typed STLC terms.

The translation from the typing rules to Datalog was straightforward, but inexact: typing is usually top-down (i.e., starting from the AST root), while generation is bottom-up (starting from the AST leaves). Accommodating the difference requires some mechanical changes during the translation.

2.2 Tracking Extra Information

The generator described in the prior subsection generates both open terms (i.e., containing free variables) and closed terms (i.e., with no free variables). It also allows for abstractions with unused parameters. For the sake of example, if we wish to only generate closed terms using all parameters, then we must track the free variables in a term. Doing so is again straightforward: we change `well-typed`'s signature to `well-typed` : `Exp` $\times$ `Type` $\times$ `Freelist`, where `Freelist` is a list of `Var`. This signature states that the given expression has the given type and contains the given free variables.

We modify the base VAR rule to be `well-typed(Var(i,t),t,[Var(i,t)])`. We modify the `abs` rule to require the parameter `Var(i,t)` to be contained in the body's free variables, and the freelist of the resulting term is the original freelist without the parameter. The APP rule leaves the freelist as-is. Our overall output is `output` : `Exp` $\times$ `Type`, where if `well-typed(e,t,[])`, then `output(e,t)`; this thus only outputs closed terms. While tracking this additional information is straightforward, it is not as simple as it may seem; we need helpers for updating tracked information (e.g., adding/removing variables to/from freelists, and checking if a variable is contained within) that entail a lot of boilerplate. To see this boilerplate, we can compare the STLC Datalog generator in Appendix A (that does not track any of this information) with the STLC Datalog generator created by compiling the accompanying artifact's Breccia STLC generator into Datalog, and note all the additional functions required to implement freelists and operations on freelists.

2.3 Bounding Generation

The generator in the prior subsection generates an unbounded number of well-typed closed terms. To ensure termination, we must make the term space finite via bounds. For STLC, unboundedness appears in variables (due to integers) and in types and terms (due to arbitrary sizes). To bound variables, VAR is modified to require that `i` must be a member of a fixed finite set. To bound types and terms, we modify all rules to enforce the term/type size to be $< k$ for some

fixed k. Varying these bounds independently from each other explores different parts of the generation space. These modifications are again straightforward, but entail yet more boilerplate.

3 E-Graphs and Equality Saturation

The Datalog generator from Section 2 creates large sets of facts representing generated terms, and they must all be present in memory simultaneously for bottom-up generation to work correctly. Since the search space contains billions of terms even after bounding, this is concerning. To address this concern, we turn to e-graphs and equality saturation. We begin with a brief overview of these two concepts and then discuss how we can make use of them in a Datalog generator.

3.1 Background

An *e-graph* is a compact structure for representing congruence classes [29]; it has been applied to automated theorem proving [34] and to the *equality saturation* [38] technique. Equality saturation starts with a set of rewrite rules from terms to equivalent terms, and a set of initial terms stored in an e-graph. From there, the current terms in the e-graph are matched against the left-hand sides of the rewrite rules, and any rules that apply are used to rewrite matched terms to new terms. The new terms are then added to the e-graph by *unifying* them with the original terms they were derived from.[5] This process repeats until a fixpoint is reached.

This idea has obvious parallels with Datalog semantics, and Egglog [42] was created to combine Datalog and equality saturation. Egglog implements the fact database using e-graphs. Egglog rules can specify in their conclusion that two terms should be considered equivalent, and the e-graph will represent this fact efficiently. Egglog conditions and rules apply to congruence classes of terms instead of individual terms, greatly speeding up the processing of equivalent terms.

3.2 Application to Generation

By defining equivalences between generated terms, we can exploit Egglog to optimize generation. Doing so is trivial: we add a single rule per equivalence relation stating that any two matching terms are equivalent. The question is what terms to make equivalent, which must be determined on a per-generator basis. For STLC and other statically-typed languages, the obvious choice is to define two terms of the same type to be equivalent. For STLC, this means adding a rule saying that if `well-typed(e1,t)` and `well-typed(e2,t)`, then `e1` $\equiv$ `e2`. Generators for other kinds of terms can define their own equivalences; for example, when generating red-black trees we could consider all trees with the same height and root color to be equivalent.

[5] Egglog's unification is a directive saying the given terms are equivalent no matter what they are, and cannot fail. This differs from Prolog's unification, which enforces equality constraints, and fails on syntactically distinct terms.

4 Breccia Design and Compilation

Datalog generators require lots of boilerplate to track required information and bound generated terms. To address this, we designed Breccia: a functional DSL that compiles to Egglog. Here we describe Breccia and how it is compiled.

4.1 The Breccia Language

We illustrate Breccia by using it to implement our running STLC example; a complete formal syntax of Breccia is in Appendix B. We begin by defining bounds as integer ranges:

```
bound TSize = 1..5
bound VarInt = 1..2
bound ESize = 1..10
```

`TSize`, `VarInt`, and `ESize` are types describing integer values within the given ranges. We next define the STLC datatypes, using mostly standard syntax. Keyword `bounded_by` specifies a bound on datatype instances, and `tracking` is used to associate some additional information with each datatype instance.

```
datatype Type bounded_by (sz: TSize) tracking () = {
  Base | Arr(Type, Type)  }

datatype Freelist = { Nil | Cons(VarInt, Type, Freelist) }

datatype Exp bounded_by (sz:ESize) tracking (fv:Freelist,ty:Type) = {
  Var(VarInt, Type) | Lam(Type, VarInt, Exp) | App(Term, Term) }
```

A `Type` is bounded by `TSize`, and does not track anything (the `tracking` keyword is still needed). Given a `Type` instance `ty`, its size can be accessed using `ty.sz`. A `Freelist` is an auxiliary (not exhaustively generated) unbounded structure which tracks no additional information because it is auxiliary. An `Exp` is bounded by `ESize` and tracks (i.e., associates) both the list of free variables in the given expression and the expression's type; given a `Exp` named `e` its tracked information can be acessed using `e.sz`, `e.fv`, and `e.ty`.

For the bounded datatypes being generated (`Type` and `Exp`), we must define how to compute bounds:

```
def Type.bound -> TSize = { self match {
  Base() => 1 | Arr(t1, t2) => 1 + t1.sz + t2.sz }}

def Exp.bound -> ESize = { self match {
  Var(_,_) => 1 | Lam(_,_,e) => 1 + e.sz |
  App(e1, e2) => 1 + e1.sz + e2.sz }}
```

Such bound definitions need only be defined once and the compiler will insert bounds checks wherever needed, i.e., whenever the generator attempts to build a `Type` or `Exp`, the compiler will insert a call to the appropriate `bound` function and check that the resulting term does not exceed the bound. We must also define helper functions to manipulate the auxiliary freelist; these are standard functional definitions for dealing with lists; see `fv_contains`, `fv_remove`, and `fv_append` in the accompanying Breccia code examples.

Now we can define the actual generators for `Type` and `Exp`. The '`#`' prepending `Type` and `Exp` allow tracked information to be accessed:

```
generator base_ty() -> #Type = { Base tracking () }

generator arr_ty(t1: #Type, t2: #Type) -> #Type = {
  Arr(t1, t2) tracking ()  }

generator var(id: VarInt, ty: #Type) -> #Exp = {
  Var(id, ty) tracking (fv: Cons(id, ty, Nil()), ty: ty) }

generator lam(ty: #Type, id: VarInt, e: #Exp) -> #Exp = {
  if (fv_contains(e.fv, id, ty)) {
    let new_fv = fv_remove(e.fv, id, ty)
    Lam(ty, id, e) tracking (fv: new_fv, ty: Arr(ty, e.ty))
  }}

generator app(e1: #Exp, e2: #Exp) -> #Exp = {
  filter e1.ty is Arr(t1, t2)
  if (t1 == e2.ty) {
    App(e1, e2) tracking (fv: fv_append(e1.fv, e2.fv), ty: t2)
  }}
```

A generator takes existing terms and bounds, and potentially checks that some criteria holds on them using `if` (e.g., `lam` and `app`) and `filter` (e.g., `app`). Helper functions are used to check and manipulate tracked information. Generators can either silently fail (resulting in no new terms), or generate a new term with some specific tracked information. Bounding is implicit, with checks inserted by the compiler based on the bounding definitions (e.g., `Type.bound` and `Exp.bound`). We must also indicate which terms should be unified, and when:

```
unify (e1 e2 : #Exp) when { e1.ty == e2.ty && e1.size == e2.size}
```

There is no main entry point to the program; generators are applied when facts matching their parameters are generated. Bounds parameters such as `VarInt` are generated automatically.

4.2 Compiling to Egglog

Breccia datatypes are translated almost one-to-one into Egglog datatypes, minus the `bounded_by` and `tracking` annotations. The `tracking` information is

compiled into a predicate specific to that datatype, associating each instance of that datatype with its tracked information; accessing tracked information via dot notation (e.g., `e.ty`) compiles into an access of the associated argument position of that predicate. Bounds declarations (e.g., `bound VarInt 1..2`) are translated into a series of facts stating that each value in the range is valid (e.g., `(VarInt 1)` and `(VarInt 2)`). Functions (`def`s) and function calls are compiled using the strategy from Pacak et al. [30], wherein return values are lifted to be parameters in Prolog style; we refer the reader to that paper for details. The `unify` directive is compiled into a rule whose conditions are derived from its body and whose conclusion unifies the relevant terms.

Each `generator` is translated into a control-flow tree (CFT) such that each non-failure execution path in the generator becomes a path from the CFT root to one of its leaves; the Breccia syntax disallows paths to converge, guaranteeing a tree. Each parameter of a `bound` type (e.g., `id : VarInt` in `lam`) is translated into a condition at the CFT root (e.g., `(VarInt id)`), ensuring that the translated Egglog rules will be instantiated with each possible value of the bound.

Internal CFT nodes consist of binding operators (`let`, `filter`, `match`) or function calls. An `if` expression is expanded to `let x = guard; match x {...}` before being translated into the CFT, where `x` is a fresh variable and `guard` is a boolean-valued expression. A binding node for some variable x to some $\langle pattern \rangle$ translates into an Egglog condition "$x = \langle pattern \rangle$". When generating conditions, we consider a single path in the CFT, thus $\langle pattern \rangle$ is only one branch of a `match` expression; `filter` is essentially a `match` with only one branch. Each function call node results in an Egglog rule per Pacak et al. [30], along with a condition on the return value; the rule also collects the conditions of all binding and call nodes along the path from the CFT root to the current call node.

CFT leaves are datatype constructor calls, i.e., newly-generated terms. Each leaf results in an Egglog rule whose conditions are collected from all binding and call nodes from the CFT root to the leaf, and whose conclusion is a variant of the newly-generated term. Leaves generating a datatype `<T>` actually conclude with a constructor call to a datatype `<T>Valid`, which has the same arguments as the `<T>` being generated. We use `<T>Valid` in order to insert bounds checks using the user-provided `T.bound` function. The relevant `T.bound` function is called with the `<T>Valid` value to see if the result is valid for that bound. If it is, then we create a `<T>`-value from the arguments of the `<T>Valid` value, i.e., we create the desired term specified by the generator.

4.3 Optimizing Compilation

A naive implementation of the compiler creates many unnecessary rules and suboptimal conditions. We apply several optimizations during compilation in order to improve the resulting Egglog program:

Eliminate Trivial Conditions. Trivially-satisfied conditions are deleted. For trivially-unsatisfied conditions we prune the containing subtree, avoiding the generation of rules that would never fire.

Copy Propagation. If a variable x is bound to e and used exactly once, then e can be inlined wherever x was used, reducing the number of variables required.

Prune Bounds Checks. If a generation function always increases a given bound (which is often the case) then we can aggressively filter out candidate generations that would definitely violate the bounds. We insert a condition that checks if the input terms have bounds such that the generated component must violate its bounds, and if so fail immediately rather than generating the new object and then rejecting it after the fact.

Collapse Independent Calls. Two function calls are *independent* if the arguments of one call do not depend on the return value of the other. Sequential independent calls in a CFT are combined into a single rule during compilation instead of making a separate rule for each of them. This optimization reduces the number of rules as well as the size of the rules associated with the later calls.

5 Evaluation

5.1 Answering the Research Questions

The two research questions posed in Section 1 were:

- **RQ1: is Datalog expressive enough to describe interesting, non-trivial constraints for test input generation?**
- **RQ2: Can e-graphs and equality saturation significantly improve the performance of a Datalog-based generator?**

To answer RQ1, we implement generators for five different statically-typed languages in Breccia: STLC [7], the simply-typed lambda calculus; System F [17, 33] (SysF) which adds parametric polymorphism to STLC; Featherweight Java [20] (FJ-cast), an object-oriented language with subtype polymorphism; an alternate version of Featherweight Java that removes casting[6] (FJ-nocast); and an alternative version of Featherweight Java that replaces subtype polymorphism with generics (FJ-gen). These languages explore a range of realistic type system features and demonstrate that Breccia can encode them all. The language implementations are available with the Breccia code.

To answer RQ2, we evaluate the generation rate of our Breccia implementations both with equality saturation ("union") and without ("no union", i.e., just Datalog), using types as equivalence classes in all cases.[7] We also implement

[6] Casting makes every type trivially inhabitable, significantly changing the space of valid progams, and making well-typed program generation far simpler.

[7] Results were obtained on an AWS EC2 m5.2xlarge instance (8 vCPUs, Intel Xeon Platinum 8175M @ 2.50GHz, 32 GB RAM) running Ubuntu 24.04 (kernel 6.14.0-1015-aws). Experiments were run 10 times, the highest and lowest results were discarded, and the remaining were averaged.

generators for all five languages in Prolog in order to get a sense of the relative performance between Prolog and Datalog generation.

Comparing the Breccia and Prolog generators directly is difficult. Breccia generates terms bottom-up in contrast to Prolog's top-down, necessitating different generation algorithms for Breccia and Prolog. We must somehow bound the state space of these algorithms to ensure a finite space of generated programs, and because these algorithms are necessarily different, different components end up needing bounds. For example, the Breccia-based generators separately bound the sizes of types and expressions, as both of these must be built up separately in a bottom-up fashion. However, the Prolog-based generators tightly couple the generation of types and expressions, necessitating only a single bound. Therefore the comparison is not "apples to apples", and is only presented to give a rough idea of their relative performance.

All generators needed multiple bounds, leading to a many-dimensional state space which is difficult to comprehensively evaluate. Within a generation strategy (Breccia or Prolog), we selected bounds such that only one specific bound was varied at a time relative to another data point; this gives a glimpse into the effect of each specific bound on the state space. We also attempted to select bounds that would explore different parts of the state space, leading to different sets of generated programs which looked relatively different.

Breccia results are shown in Table 1 (STLC and SysF), Table 2 (FJ-cast and FJ-nocast), and Table 3 (FJ-gen); we use tables instead of graphs as the results vary wildly in magnitude. The generation rates for STLC and SysF (Table 1) reach as high as billions of terms per second. On the slower end, FJ-cast and FJ-nocast (Table 2) are generally dozens per second, with FJ-gen in between. In all cases, Datalog successfully generates many terms with complex invariants, and does so at least sufficiently fast for fuzzing, leading us to conclude that RQ1 is true. Furthermore, in all cases, equality saturation improves the generation rate, with STLC and SysF (Table 1) seeing even orders of magnitude improvements; this leads us to conclude that RQ2 is also true. Altogether, generation rates and equality saturation's effectiveness vary significantly. There is another surprising observation: within STLC union, bound (6-2-9) is notably slower than others; the relevant row is highlighted in Table 1. The next row shows that the greater STLC bound (6-2-10) is faster, as is SysF, despite these being more complex. To explain why, we also look at generation rate over time (Figure 4).

Figure 4 takes a representative sample of languages and bounds, and plots their generation rate over time. Time is on the x-axis, the number of generated terms is on the y-axis, and both are log scale. The log scale is because, in an unbounded space, generators produce exponentially more terms as the bound linearly increases; this follows directly from terms being recursively defined. At log scale, we thus expect to see straight lines for unbounded spaces. The lines for STLC (6-2-15) and SysF (2-3-2-10) roughly follow this pattern, while the others all plateau at some point, some faster than others. In particular, STLC (6-2-9) and FJ-cast plateau quickly, and FJ-gen also plateaus but later than others.

Table 1: Breccia generation rates (#terms/s) for STLC and SysF; the other columns specify the bounds used. Bound (6-2-9) for STLC union is highlighted as it is unusually slow compared to the other STLC union results.

type size	# vars	expr size	STLC union	STLC no union	SysF # type vars	SysF union	SysF no union
3	2	10	12.7m	132.1k	2	94.7b	414.8k
4	2	14	1.9b	536.3k	2	2996.6b	335.2k
5	2	10	2.5m	209.0k	1	2.7b	258.2k
6	2	9	415.4k	129.9k	2	89.1m	124.2k
6	2	10	2.5m	210.6k	5	192.9k	13.2k
6	2	12	55.4m	309.3k	2	391.2m	276.4k
6	2	15	4.5b	236.0k	2	564.5m	302.2k
6	5	10	168.8m	168.6k	2	85.4m	39.7k
9	2	10	3.4m	518.3k	2	3.3k	2.9k

Table 2: Breccia generation rates (#terms/s) for FJ-nocast and FJ-cast; the other columns specify the bounds used.

language	# classes	# fields	class ids	var ids	expr size	num meth	# args	# vars	class depth	union rate	no union rate
FJ-nocast	2	2	2	2	5	6	2	2	2	14.5	9.2
	2	2	2	2	6	1	2	2	2	54.7	17.0
	2	2	2	2	9	1	2	2	2	53.6	29.0
	2	2	2	2	12	1	2	2	2	53.3	28.7
	2	6	2	2	5	2	2	2	2	113.4	21.3
	3	2	2	2	6	2	2	2	1	172.6	11.3
	3	2	2	2	6	2	2	2	3	113.2	21.8
	3	2	4	4	7	2	3	4	2	17.4	9.1
	4	3	3	3	6	3	4	3	3	16.7	14.8
	6	2	3	2	5	2	2	2	3	74.0	16.8
FJ-cast	2	2	2	2	5	6	2	2	2	36.1	31.8
	2	2	2	2	6	1	2	2	2	81.2	73.5
	2	2	2	2	9	1	2	2	2	82.3	71.2
	2	2	2	2	12	1	2	2	2	76.8	73.4
	2	6	2	2	5	2	2	2	2	53.5	48.7
	3	2	2	2	6	2	2	2	1	81.3	71.9
	3	2	2	2	6	2	2	2	3	52.0	49.9
	3	2	4	4	7	2	3	4	2	76.1	78.6
	4	3	3	3	6	3	4	3	3	85.3	44.1
	6	2	3	2	5	2	2	2	3	106.7	53.9

Table 3: Breccia generation rates (#terms/s) for FJ-gen; the other columns specify the bounds used.

# classes	# fields	class ids	var ids	expr size	num meth	# args	# vars	class depth	# class vars	union rate	no union rate
2	2	2	2	5	6	2	2	2	2	76.5	35.0
2	2	2	2	6	1	2	2	2	2	2.5k	379.1
2	2	2	2	9	1	2	2	2	2	2.5k	384.9
2	2	2	2	12	1	2	2	2	2	2.5k	379.7
2	6	2	2	5	2	2	2	2	2	23.5	55.8
3	2	2	2	6	2	2	2	1	2	70.8	177.9
3	2	2	2	6	2	2	2	3	2	19.7	5.8
3	2	4	4	7	2	3	4	2	2	87.7	30.2
4	3	3	3	6	3	4	3	3	2	103.0	35.8
6	2	3	2	5	2	2	2	3	2	51.8	19.8

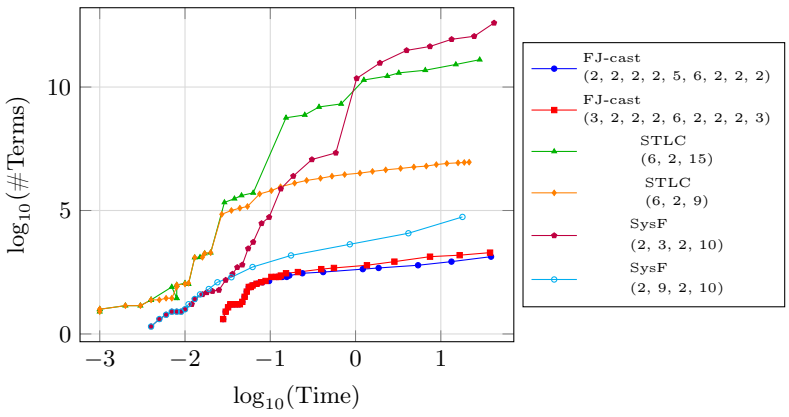

Fig. 4: Log Time (seconds) vs Log #Terms

Multiple bounds are used for each language, describing a multi-dimensional space. We hypothesize that the bounds are interacting with generation in interesting and unpredictable ways. In particular, we may reach a point where most existing terms, if used as components to generate a new term, will cause the new term to violate the bounds and be discarded; thus only a small number of valid new terms may be generated at each step, greatly slowing the generation rate. This would at least partially explain the poor performance of the FJ generators vs. STLC, as by having many more bounding parameters, the FJ generators would have many more opportunities to hit this issue. If and when this happens, and whether the slow-down is gradual or quick, is a function of the bounds and the space being explored. Characterizing this behavior is an interesting question to explore in future work.

We also briefly look at the generation rates for Prolog, shown in Table 4 (STLC and SysF), Table 5 (FJ-cast and FJ-nocast), and Table 6 (FJ-gen). Prolog, due to its top-down generation strategy, outputs terms and then discards them, whereas Datalog stores all generated terms in memory. As a point of interest, we investigated how much of the Prolog generation time was due to specifically to I/O, and include those numbers in the tables. That is, we measure both raw generation rate, as well as generation plus the cost to write programs to disk We see that Prolog generation is consistently orders of magnitude faster for FJ-nocast, FJ-cast, and FJ-gen (Tables 5 and 6 compared to Table 2), but orders of magnitude slower for STLC and SysF (Table 4 compared to Table 1)—however, we again must be wary of drawing conclusions due to the inherently different generation approach Prolog uses. The I/O cost of writing out terms is very large, but unavoidable in fuzzing using Prolog. The variance in generation rate across the Prolog generators is much lower than it is for Breccia; we hypothesize this is because bounding issues are a well-known issue with Prolog generation, and we specifically bounded the Prolog code so as to mitigate this issue.

Table 4: Prolog generation rates (#terms/s) for STLC and SysF; the other columns specify the bounds used. Bounds (4-3) and (5-3) were not measured for SysF.

init bound	# vars	STLC no IO	STLC with IO	SysF no IO	SysF with IO
4	3	357.9k	175.1k	/	/
5	3	351.6k	162.5k	/	/
5	5	354.3k	163.3k	99.6k	60.9k
6	3	346.9k	152.0k	94.4k	54.0k
7	7	340.5k	141.7k	85.9k	53.1k
9	3	328.0k	126.0k	71.6k	42.8k
9	9	323.4k	125.7k	76.4k	47.4k
11	11	312.1k	112.9k	69.8k	43.6k
13	3	305.3k	103.5k	53.8k	33.8k

Table 5: Prolog generation rates (#terms/s) for FJ-nocast and FJ-cast; the other columns specify the bounds used.

language	# classes	# fields	num meth	# vars	term bound	no IO rate	with IO rate
FJ-nocast	2	4	2	4	3	365.7k	16.0k
	2	4	2	4	5	290.3k	14.6k
	2	4	2	4	8	314.9k	13.1k
	2	4	2	4	12	307.1k	11.6k
	2	4	2	12	5	378.6k	7.6k
	2	4	6	4	5	324.0k	7.5k
	2	12	2	4	5	305.9k	8.4k
	4	6	4	6	5	261.2k	4.1k
	6	4	2	4	5	128.0k	6.0k
FJ-cast	2	4	2	4	3	352.3k	15.1k
	2	4	2	4	5	307.1k	14.8k
	2	4	2	4	8	287.3k	12.5k
	2	4	2	4	12	284.0k	11.5k
	2	4	2	12	5	370.9k	8.7k
	2	4	6	4	5	295.5k	7.3k
	2	12	2	4	5	313.1k	8.5k
	4	6	4	6	5	197.9k	4.3k
	6	4	2	4	5	97.1k	5.5k

Table 6: Prolog generation rates (#terms/s) for FJ-gen; the other columns specify the bounds used.

# classes	# fields	num meth	NumParams	# type vars	ExpBound	TypeBound	no IO rate	with IO rate
2	2	5	1	1	3	6	71.0k	9.7k
3	2	2	1	1	3	6	83.8k	12.0k
3	2	5	1	1	3	6	59.5k	7.7k
3	2	5	1	1	3	8	62.0k	7.9k
3	4	5	1	1	3	6	60.9k	7.3k

5.2 A Mini-Fuzzing Campaign

To investigate the bug-finding capability of our generators we ran a mini-fuzzing campaign for two of our example languages: we chose STLC as a representative of functional languages and FJ-nocast as a representative of OOP languages. For STLC we implemented an interpreter, and for FJ-nocast we implemented a compiler (both with a type checker in the front-end). For each implementation we manually created a number of mutants, each with a different injected fault. We then selected a set of bounds and used the generated terms from both the Prolog and Breccia generators to fuzz the implementations, tracking how many of the injected faults were found. The results are shown in Table 7. We see that both the Prolog and Breccia generators find roughly the same number of bugs (the differences are because, again, the bounding strategies are different between Prolog and Breccia and so they aren't exploring exactly the same space of programs).

Table 7: Fuzzing Results.

Language Fuzzed	**Generator w/ bounds**	**#Faults found / Total faults**
STLC	Breccia (7,3,2)	6 / 7
	Prolog (3,3)	7 / 7
FJ-nocast	Breccia (2,2,1,0,2,0,0,0,2)	14 / 14
	Prolog (2,1,1,1,1)	13 / 14

6 Limitations and Threats to Validity

All of the benchmarks presented in Section 5 deal with programs which are inherently compositional in nature (e.g., expressions contain subexpressions). We have not evaluated Breccia on non-compositional problems. Breccia's compositional nature is best suited to problems wherein it is straightforward to determine if a larger term is valid given some previously-generated valid subterms. Problems requiring state outside of a local group of terms are more challenging to encode, though not impossible, and likely have implications for performance.

Our evaluation does not consider dynamically-typed languages, though Breccia is applicable to such languages— ASTs are inherently compositional, and so syntactically valid programs in any language is doable in Breccia. Prior work using Prolog has generated JavaScript using a rudimentary type system to avoid uninteresting behavior [14], and Breccia can be used in the same way.

The bounds chosen in our evaluation were chosen to explore different parts of the state space of possible programs. However, we did not comprehensively search the space of possible bounds, making these choices somewhat arbitrary.

7 Related Work

7.1 Fuzzing using Logic Programming Variants

Both Grygiel et al. [18] and Tarau [37] use Prolog to generate well-typed terms in the simply-typed lambda calculus [7]. However, this was for the purpose of studying properties of the terms and distributions over them, as opposed to using them for testing purposes. Dewey et al. [14] introduce the idea of using Prolog to generate programs for fuzzing. They later apply this idea to fuzzing Rust [15], student assignments [11], and a custom Scala-inspired language [10], as well as generating complex data structures with non-trivial invariants [13]. However, multiple works argue that Prolog-based fuzzing is too difficult to use, too limited, or too sensitive to optimizations to be widely applicable [3, 6, 19].

Luck [23] is a DSL for property-based testing [8], which has a semantics combining top-down Prolog-style backtracking with constraint solving. This semantics is exposed to the user through a functional interface, and users can work with concrete and symbolic data. Breccia, in contrast, performs bottom-up generation with equality saturation. Breccia's bottom-up nature imparts a very different approach to writing generators, and presents an arguably simpler interface, since user-exposed data is always concrete.

7.2 Functional Languages and Datalog

Pacak et al. [31] define a DSL for describing type systems which compiles to a Datalog-based incremental typechecker. Pacak et al. [30] later describe shortcomings in Datalog as a user-level programming language and use those shortcomings to motivate Functional IncA, a functional programming language with algebraic data types and sets that compiles to Datalog. Neither approach can be used for generating inputs based on constraints (including type systems). Other works, such as Flix [24,25], Datafun [2], Formulog [4], and Flan [1] either extend Datalog with functional language constructs or embed Datalog inside a functional language, usually designed for a specific purpose such as program analysis. Again, none of these languages support generating inputs based on constraints.

7.3 Equality Saturation and Egglog

Willsey et al. introduce Egg [40], a modern e-graph library for equality saturation. Zhang et al. introduce Egglog [42], which unifies Datalog and equality saturation into a single fixpoint reasoning system. Equality saturation has been applied across many domains [9, 16, 28, 32, 39, 41]. None of these works apply equality saturation to the problem of input generation.

References

1. Abeysinghe, S., Xhebraj, A., Rompf, T.: Flan: An Expressive and Efficient Datalog Compiler for Program Analysis. Proceedings of the ACM on Programming Languages **8**(POPL), 2577–2609 (2024)

2. Arntzenius, M., Krishnaswami, N.R.: Datafun: a functional Datalog. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming. pp. 214–227 (2016)
3. Awar, N.A., Jain, K., Rossbach, C.J., Gligoric, M.: Programming and execution models for parallel bounded exhaustive testing. Proc. ACM Program. Lang. **5**(OOPSLA) (oct 2021). https://doi.org/10.1145/3485543, https://doi-org.libproxy.csun.edu/10.1145/3485543
4. Bembenek, A., Greenberg, M., Chong, S.: Formulog: Datalog for SMT-based static analysis. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–31 (2020)
5. Ceri, S., Gottlob, G., Tanca, L., et al.: What you always wanted to know about Datalog(and never dared to ask). IEEE transactions on knowledge and data engineering **1**(1), 146–166 (1989)
6. Chaliasos, S., Sotiropoulos, T., Spinellis, D., Gervais, A., Livshits, B., Mitropoulos, D.: Finding typing compiler bugs. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 183–198. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523427, https://doi.org/10.1145/3519939.3523427
7. Church, A.: A formulation of the simple theory of types. The Journal of Symbolic Logic **5**(2), 56–68 (1940), http://www.jstor.org/stable/2266170
8. Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 268–279. ICFP '00, ACM, New York, NY, USA (2000). https://doi.org/10.1145/351240.351266, http://doi.acm.org/10.1145/351240.351266
9. Coward, S., Constantinides, G.A., Drane, T.: Automating Constraint-Aware Datapath Optimization Using E-Graphs. In: 2023 60th ACM/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2023)
10. Dewey, K.: Automated Black Box Generation of Structured Inputs for Use in Software Testing, Chapter 7. Ph.D. thesis, University of California, Santa Barbara, USA (2017), http://www.escholarship.org/uc/item/9362s6mg
11. Dewey, K., Conrad, P., Craig, M., Morozova, E.: Evaluating Test Suite Effectiveness and Assessing Student Code via Constraint Logic Programming. In: Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education. pp. 317–322. ITiCSE '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3059009.3059051, http://doi.acm.org/10.1145/3059009.3059051
12. Dewey, K., Hairapetian, S., Gavrilov, M.: MiMIs: Simple, Efficient, and Fast Bounded-Exhaustive Test Case Generators. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). pp. 51–62 (2020). https://doi.org/10.1109/ICST46399.2020.00016
13. Dewey, K., Nichols, L., Hardekopf, B.: Automated Data Structure Generation: Refuting Common Wisdom. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1. pp. 32–43. ICSE '15, IEEE Press, Piscataway, NJ, USA (2015), http://dl.acm.org/citation.cfm?id=2818754.2818761
14. Dewey, K., Roesch, J., Hardekopf, B.: Language Fuzzing Using Constraint Logic Programming. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 725–730. ASE '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2642937.2642963, http://doi.acm.org/10.1145/2642937.2642963

15. Dewey, K., Roesch, J., Hardekopf, B.: Fuzzing the Rust Typechecker Using CLP. In: Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 482–493. ASE '15, IEEE Computer Society, Washington, DC, USA (2015). https://doi.org/10.1109/ASE.2015.65, http://dx.doi.org/10.1109/ASE.2015.65
16. Fallin, C.: Cranelift RFC: Middle-end optimization framework based on e-graphs (2022), https://github.com/bytecodealliance/rfcs/blob/main/accepted/cranelift-egraph.md, bytecode Alliance RFC
17. Girard, J.Y.: Une extension de L'interpretation de gödel a L'analyse, et son application a L'elimination des coupures dans L'analyse et la theorie des types. In: Fenstad, J. (ed.) Proceedings of the Second Scandinavian Logic Symposium, Studies in Logic and the Foundations of Mathematics, vol. 63, pp. 63–92. Elsevier (1971). https://doi.org/https://doi.org/10.1016/S0049-237X(08)70843-7, https://www.sciencedirect.com/science/article/pii/S0049237X08708437
18. Grygiel, K., Lescanne, P.: Counting and generating lambda terms. CoRR **abs/1210.2610** (2012), http://arxiv.org/abs/1210.2610
19. Hyatt, S.C., Dewey, K.: Mutation-Based Fuzzing of the Swift Compiler with Incomplete Type Information. In: 2025 IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 58–68 (2025). https://doi.org/10.1109/ICST62969.2025.10989032
20. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. In: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 132–146. OOPSLA '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/320384.320395, https://doi.org/10.1145/320384.320395
21. Jackson, D., Damon, C.A.: Elements of style: analyzing a software design feature with a counterexample detector. SIGSOFT Softw. Eng. Notes **21**(3), 239–249 (May 1996). https://doi.org/10.1145/226295.226322, https://doi.org/10.1145/226295.226322
22. Kuraj, I., Kuncak, V., Jackson, D.: Programming with Enumerable Sets of Structures. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 37–56. OOPSLA 2015, ACM, New York, NY, USA (2015). https://doi.org/10.1145/2814270.2814323, http://doi.acm.org/10.1145/2814270.2814323
23. Lampropoulos, L., Gallois-Wong, D., Hriţcu, C., Hughes, J., Pierce, B.C., Xia, L.y.: Beginner's luck: a language for property-based generators. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 114–129. POPL '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009868, https://doi.org/10.1145/3009837.3009868
24. Madsen, M., Lhoták, O.: Fixpoints for the masses: programming with first-class Datalog constraints. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–28 (2020)
25. Madsen, M., Yee, M.H., Lhoták, O.: From Datalog to flix: A declarative language for fixed points on lattices. ACM SIGPLAN Notices **51**(6), 194–208 (2016)
26. Marinov, D., Khurshid, S.: TestEra: A Novel Framework for Automated Testing of Java Programs. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering. pp. 22–. ASE '01, IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=872023.872551

27. Miller, B.P., Fredriksen, L., So, B.: An Empirical Study of the Reliability of UNIX Utilities. Commun. ACM **33**(12), 32–44 (Dec 1990). https://doi.org/10.1145/96267.96279, http://doi.acm.org/10.1145/96267.96279
28. Nandi, C., Willsey, M., Anderson, A., Wilcox, J.R., Darulova, E., Grossman, D., Tatlock, Z.: Synthesizing structured CAD models with equality saturation and inverse transformations. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 31–44 (2020)
29. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. Journal of the ACM (JACM) **27**(2), 356–364 (1980)
30. Pacak, A., Erdweg, S.: Functional Programming with Datalog. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming (ECOOP 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 222, pp. 7:1–7:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.ECOOP.2022.7, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2022.7
31. Pacak, A., Erdweg, S., Szabó, T.: A systematic approach to deriving incremental type checkers. Proceedings of the ACM on Programming Languages **4**(OOPSLA), 1–28 (2020)
32. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. p. 1–11. PLDI '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2737924.2737959, https://doi.org/10.1145/2737924.2737959
33. Reynolds, J.C.: Towards a theory of type structure. In: Robinet, B. (ed.) Programming Symposium. pp. 408–425. Springer Berlin Heidelberg, Berlin, Heidelberg (1974)
34. Rümmer, P.: E-matching with free variables. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning. pp. 359–374. Springer (2012)
35. Senni, V., Fioravanti, F.: Generation of test data structures using constraint logic programming. In: Proceedings of the 6th international conference on Tests and Proofs. pp. 115–131. TAP'12, Springer-Verlag, Berlin, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30473-6_10, http://dx.doi.org/10.1007/978-3-642-30473-6_10
36. Takashima, Y., Martins, R., Jia, L., Păsăreanu, C.S.: SyRust: automatic testing of Rust libraries with semantic-aware program synthesis. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 899–913. PLDI 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3453483.3454084, https://doi.org/10.1145/3453483.3454084
37. Tarau, P.: On Type-Directed Generation of Lambda Terms. In: Technical Communications of the 31st International Conference on Logic Programming, ICLP 2015, August 31 - September 4, 2015, Cork, Ireland (2015)
38. Tate, R., Stepp, M., Tatlock, Z., Lerner, S.: Equality saturation: a new approach to optimization. In: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 264–276 (2009)
39. VanHattum, A., Nigam, R., Lee, V.T., Bornholt, J., Sampson, A.: Vectorization for digital signal processors via equality saturation. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 874–886 (2021)

40. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and extensible equality saturation. Proceedings of the ACM on Programming Languages **5**(POPL), 1–29 (2021)
41. Yang, Y., Phothilimthana, P., Wang, Y., Willsey, M., Roy, S., Pienaar, J.: Equality Saturation for Tensor Graph Superoptimization. In: Smola, A., Dimakis, A., Stoica, I. (eds.) Proceedings of Machine Learning and Systems. vol. 3, pp. 255–268 (2021), https://proceedings.mlsys.org/paper_files/paper/2021/file/cc427d934a7f6c0663e5923f49eba531-Paper.pdf
42. Zhang, Y., Wang, Y.R., Flatt, O., Cao, D., Zucker, P., Rosenthal, E., Tatlock, Z., Willsey, M.: Better Together: Unifying Datalog and Equality Saturation. Proceedings of the ACM on Programming Languages **7**(PLDI), 468–492 (2023)

A Datalog Generator for Simply-Typed λ-Calculus

```
;; Simply-typed lambda calculus in egglog

;; Define types in STLC (Figure 2)
(datatype type
  (Base)                ;; Base type
  (Arr type type))      ;; Function type: t1 -> t2

;; Define expressions in STLC (Figure 3)
(datatype exp
  (Var i64 type)        ;; Variables: Var(Int, Type)
  (Abs exp exp)         ;; Lambda abstraction: Abs(Var, Exp)
  (App exp exp))        ;; Application: App(Exp, Exp)

;; Predicates for well-formedness and well-typedness
(relation type (type i64))           ;; type : Type, with size
(relation welltyped (exp type i64))  ;; well-typed : Exp x Type, with size

;; Bounds
(relation maxsize (i64))
(relation maxidx (i64))
(relation valididx (i64))

(maxsize 4)
(maxidx 1)

;; Generate valid variable indices
(valididx 1)
(rule
  ((valididx i) (maxidx m) (> m i))
  ((valididx (+ i 1))))

;; Base case: Base is a well-formed type
```

```
(type (Base) 1)

;; Inductive case: if type(t1) and type(t2), then type(Arr(t1,t2))
(rule
  ((maxsize m)
   (type t1 s1)
   (type t2 s2)
   (> m (+ s1 s2)))
  ((type (Arr t1 t2) (+ 1 (+ s1 s2)))))

;; Base case for var: if i is an integer and type(t),
;; then welltyped(Var(i,t), t)
(rule
  ((type t s) (valididx i))
  ((welltyped (Var i t) t 1)))

;; Rule for abs: if welltyped(Var(i,t1),t1) and welltyped(e,t2),
;; then welltyped(Abs(Var(i,t1),e), Arr(t1,t2))
(rule
  ((welltyped (Var i t1) t1 s1)
   (welltyped e t2 s2)
   (maxsize m)
   (> m (+ s1 s2)))
  ((welltyped (Abs (Var i t1) e) (Arr t1 t2) (+ 1 (+ s1 s2)))))

;; Rule for app: if welltyped(e1,Arr(t1,t2)) and welltyped(e2,t1),
;; then welltyped(App(e1,e2),t2)
(rule
  ((welltyped e1 (Arr t1 t2) s1)
   (welltyped e2 t1 s2)
   (maxsize m)
   (> m (+ s1 s2)))
  ((welltyped (App e1 e2) t2 (+ 1 (+ s1 s2)))))

;; Equality saturation: two terms of the same type are equivalent
(rule
  ((welltyped e1 t s1) (welltyped e2 t s2))
  ((union e1 e2)))

(run 10000)
(print-function welltyped 100000)
(print-size welltyped)
```

B Breccia Syntax

```
Program         ::= Declaration*

Declaration     ::= BoundDecl | DatatypeDecl | GenTypeDecl | NormalFuncDecl
                  | BoundFuncDecl | GenFuncDecl

BoundDecl       ::= bound Id = Integer .. =? Integer

DatatypeDecl    ::= datatype Id = { Variant+ }

GenTypeDecl     ::= datatype Id  bounded_by ( Param* ) \
                    tracking ( Param* ) = { Variant+ }

Variant         ::= | Id [( TypeList )]
TypeList        ::= Type [, Type]*

NormalFuncDecl  ::= def Id ( FuncParams ) -> Type = { Block }

BoundFuncDecl   ::= def Id.bound -> Id = { Block }

GenFuncDecl     ::= generator Id ( FuncParams ) -> # Id = { Block }

FuncParams      ::= [Param [, Param]*]
Param           ::= Id : TypeRef
TypeRef         ::= Type | # Type

Type            ::= Int | Bool | Id

Block           ::= Statement* Expr

Statement       ::= LetStmt | FilterStmt

LetStmt         ::= let Id = Expr Newline

FilterStmt      ::= filter Expr is Pattern Newline

Expr            ::= IfExpr | MatchExpr | GenReturnExpr | BinaryExpr | Term

IfExpr          ::= if ( Condition ) { Block } [else { Block }]

MatchExpr       ::= Expr match { MatchCase* }

MatchCase       ::= | Pattern => Expr

GenReturnExpr   ::= Id [( Args )] tracking ( Id : Expr [, Id : Expr]* )
```

```
BinaryExpr      ::= Term Operator Term

Term            ::= Id | Integer | true | false | fail | FuncCall
                  | ConstructorCall | FieldAccess | ( Expr )

FuncCall        ::= Id ( Args )
ConstructorCall::= Id ( Args )
FieldAccess     ::= Id . Id
Args            ::= [Expr [, Expr ]*]

Pattern         ::= Id ( [PatternList] ) | Id | _

PatternList     ::= Pattern [, Pattern]*

Condition       ::= Expr CompOperator Expr
Operator        ::= + | - | || | &&
CompOperator    ::= == | != | < | > | <= | >=
```