2015 30th IEEE/ACM International Conference on Automated Software Engineering

Fuzzing the Rust Typechecker Using CLP

Kyle Dewey Jared Roesch Ben Hardekopf
University of California, Santa Barbara
{kyledewey, jroesch, benh}@cs.ucsb.edu

***Abstract*—Language fuzzing is a bug-finding technique for testing compilers and interpreters; its effectiveness depends upon the ability to automatically generate valid programs in the language under test. Despite the proven success of language fuzzing, there is a severe lack of tool support for fuzzing statically-typed languages with advanced type systems because existing fuzzing techniques cannot effectively and automatically generate well-typed programs that use sophisticated types. In this work we describe how to automatically generate well-typed programs that use sophisticated type systems by phrasing the problem of well-typed program generation in terms of Constraint Logic Programming (CLP). In addition, we describe how to specifically target the typechecker implementation for testing, unlike all existing work which ignores the typechecker. We focus on typechecker precision bugs, soundness bugs, and consistency bugs. We apply our techniques to Rust, a complex, industrial-strength language with a sophisticated type system.**

I. Introduction

The central idea of a language fuzzer is to automatically generate valid programs in a given language, which are then fed to a language implementation under test in order to check for crashes or miscompilations. This idea is well-established as a confidence-building and bug-finding technique for compilers and interpreters; for example, thousands of bugs have collectively been found by the jsfunfuzz [1], LangFuzz [2], and CSmith [3] language fuzzers.

However, no existing language fuzzers target any statically-typed languages with advanced type systems that include features such as parametric or subtype polymorphism, generics, pattern matching, type classes, etc—that is, languages such as Java, C♯, ML, Haskell, Scala, Swift, and Rust. The reason behind this lack of tool support is that current language fuzzing techniques are unable to effectively and automatically generate well-typed programs in such languages. Moreover, even for languages with simple type systems that *can* be fuzzed using current techniques, the typechecker is viewed merely as an obstacle that must be overcome in order to test the rest of the language implementation, rather than a component worthy of being tested for its own sake. The typechecker is responsible for enforcing guaranteed program behaviors such as memory safety; the more complex a language's type system is, the more important it becomes that the typechecker itself is tested for correctness. Current language fuzzing methods do not address the necessary techniques and methodology for testing a typechecker.

In this paper we advance the state of the art in language fuzzing in several ways. First, we take advantage of the well-known idea of *propositions as types* and *programs as proofs* [4] (also known as the *Curry-Howard Correspondance*) in order to phrase the problem of well-typed program generation as a constraint satisfaction problem expressible in constraint logic programming (CLP), e.g., Prolog [5], [6]. Because a type is a logical proposition, we can straightforwardly encode types and type systems using CLP. Because programs are proofs, querying the CLP engine whether a type is "true" corresponds to generating a well-typed program. The nondeterminism inherent in CLP languages means that when there are multiple possible proofs (i.e., multiple well-typed programs) the CLP engine can easily generate all possible solutions—that is, it can output as many well-typed programs as we desire. This method for automated program generation allows us to take advantage of long-standing existing implementations of CLP [7], [8] and community wisdom about effectively using CLP [9].

Our second advance describes techniques for specifically testing typechecker implementations. The three main kinds of typechecker bugs we target are (1) **precision bugs**, where the typechecker conservatively rejects well-behaved programs it should accept; (2) **soundness bugs**, where the typechecker optimistically accepts potentially ill-behaved programs it should reject; and (3) **consistency bugs**, where the typechecker treats a set of equivalent programs (in terms of being well- or ill-typed) inconsistently, accepting some while rejecting others.

Testing for precision bugs requires only that we generate well-typed programs as described previously and then determine whether the typechecker erroneously rejects them. Testing for soundness bugs requires that we generate *ill-typed* programs to see whether the typechecker erroneously accepts them. However, merely generating arbitrarily ill-typed programs is trivial and mostly ineffective; the trick is to generate programs that are *non-obviously* ill-typed, i.e., ones that could conceivably be accepted even by a typechecker that has been carefully implemented and reviewed. In order to accomplish this feat, we describe a principled and automatic technique for generating "almost well-typed" programs that builds on our previous CLP-based technique for generating well-typed programs. Testing for consistency bugs requires that we generate equivalence classes of programs that are all well-typed or ill-typed in the same way and then determine whether the typechecker accepts or rejects all of them similarly. We use a set of simple code transformations in CLP that allow us to generate such type-equivalent programs.

To demonstrate the practical utility of our techniques, we apply them to testing the Rust language [10] typechecker implementation. Rust is a statically-typed language with an advanced type system that is being actively developed by Mozilla. Rust's type system serves as an excellent case study of our techniques, as it is highly sophisticated, lacks a formal specification, and is under constant modification. These properties are endemic to real, industry-strength languages, and tackling them head-on allows us to push our own techniques to their limits. While the lack of a formal type system specification prevents us from establishing ground truth regarding what is well-typed, it serves to open a dialog with the language

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.65

developers regarding the implications of the type system and typechecker implementation decisions they make. That is, our work can be used to find oddities and problems with a type system under development, and these issues can be fed through the development cycle to allow for further type system refinement as well as typechecker fixes. We have worked closely with the Rust development team during this case study, and our efforts have raised a number of questions that the Rust developers have had to debate and think hard about in order to decide what is and is not correct behavior.

Overall, we make the following contributions in this work:

- An approach for generating well-typed programs in statically-typed languages with advanced type systems. (Section III)
- A technique for testing the precision of typechecker implementations, based on the approach described above. (Section IV-A)
- A technique for testing the soundness of typechecker implementations, based on our notion of "almost well-typed" programs. (Section IV-B)
- A technique for testing the consistency of typechecker implementations, based on our notion of "type-equivalent" programs. (Section IV-C)
- The application of all the above techniques towards fuzzing the Rust language typechecker implementation, along with an evaluation and discussion of the results. (Sections V and VI)

II. Related Work

We discuss existing work relevant to language fuzzing. None of the language fuzzers discussed below, even the ones targeting statically-typed languages, attempt to test the typechecker itself.

The most common approach to language fuzzing employs *stochastic grammars* [11], which perform a random walk over a context-free grammar according to some probability distribution in order to generate syntactically valid programs. There are many existing fuzzers based on this technique, including `jsfunfuzz` [1], `cross_fuzz` [12], and `arithfuzz` [13]. The advantage of the stochastic grammar technique is that it is both simple and language-agnostic, a property exploited by LangFuzz [2] to test both JavaScript and PHP. The downside is that this technique implicitly assumes that syntactic validity is the only important property for test programs to have, as program generation constraints are based solely on the language grammar. The stochastic grammar approach is not well-suited to statically-typed languages with non-trivial type systems, as the probability of randomly generating well-typed programs is typically extremely low.

There has been a significant amount of work that attempts to adapt stochastic grammars to fuzz statically-typed languages. One idea is to restructure the language grammars to incorporate type system information directly into the grammar and thus emit only well-typed terms, as seen in McKeeman [11] and St-Amour et al. [14]. This approach necessarily gives up on certain kinds of programs where the type information is too complex to express in a grammar, to the point where it is unable to express most valid programs as the language and type system under test become more and more complex. A second idea is to generate syntactically valid programs and then filter out ill-typed programs post hoc, as in Gligoric et al. [15]. The effectiveness of this approach is dependent on the relative frequencies with which well-typed and ill-typed programs are generated, and experience has shown that the number of well-typed programs is usually dwarfed by the number of ill-typed programs. A third idea is to augment the stochastic grammar to integrate additional checks and analyses, as done for CSmith [3]. This approach, however, is very specific to the language under test and can be extremely complex to implement and maintain. Overall, there are many hurdles to overcome in order to make stochastic grammars emit well-typed programs, as the technique fundamentally does not concern itself with type information.

A simpler approach is to develop a program generation strategy with a built-in knowledge of type systems, enabling the direct generation of well-typed programs. For example, both Eclat [16] and JCrasher [17] generate well-typed Java programs by design. However, these techniques are still problematic from the standpoint of testing statically-typed languages in general, as both generation strategies are highly specific to testing Java code. Moreover, they are limited in several ways. First, they must take in existing classes and inheritance hierarchies as input and cannot generate new classes and inheritance hierarchies. Second, they are fundamentally incapable of handling generics, parametric polymorphism, and higher-order functions; these are inherent limitations of the underlying technique. These restrictions are all side-effects of the way in which generation proceeds, as the underlying generation algorithm is specific to the subset of Java that was chosen to be generated. Overall, while Eclat and JCrasher are capable of generating well-typed programs, their generation algorithms lack expressiveness and generality.

Fetscher et al. [18] discuss a more general approach for well-typed program generation that can be applied to arbitrary languages, though their focus is on testing semantic properties of a formal language definition rather than fuzzing a language implementation. The central idea is to model a language's type system using PLT Redex [19], and then apply a custom-built constraint solver to generate programs which are well-typed according to the given model. While this approach is more general than Eclat [16] or JCrasher [17], it is subject to technical problems which practically limit its usage to simple type systems lacking advanced features like parametric polymorphism. In contrast, our CLP-based technique reuses existing high-performance CLP engines (e.g., [7], [8]), and can easily handle more advanced type system features. Most importantly, we describe techniques to generate type-equivalent programs and programs which are ill-typed in subtle ways, whereas Fetscher et al. only describes the generation of well-typed programs. Overall, our focus is on testing the typechecker itself, whereas Fetscher et al. is concerned only with getting programs past the typechecker.

Another general approach is seen in program synthesis (e.g. [20], [21]), wherein programs with highly specific behaviors are automatically constructed, usually with the help of pre-existing SMT solvers [22], [23]. While synthesis techniques are certainly applicable to fuzzing statically-typed languages,

they tend to be prohibitively complex and computationally expensive for this purpose. Synthesis problems often involve many constraints from different domains, whereas most type systems (e.g., those described in Pierce [24]) require only relatively simple equality constraints. Moreover, typing rules are often written in the style of inductive inference rules, which SMT solvers cannot handle without some additional translation [25]. Overall, the concern with fuzzing is to generate programs with relatively few constraints as quickly as they can be tested, whereas with synthesis the interest is in generating some program which satisfies some high complexity constraints. While tens of seconds per program may be considered acceptable or even fast for a synthesis problem (e.g., [21]), this is impractically slow for fuzzing purposes. As such, program synthesis is not a very applicable way to view the fuzzing problem.

In our own prior work [26], we showed how to use CLP to fuzz dynamically-typed languages with special focus on JavaScript. The paper briefly mentions a very simple, unsound type system for JavaScript designed to avoid common runtime errors. In this work, we focus specifically on testing typecheckers for statically-typed languages with advanced type systems. While at a high level both that paper and this one use CLP to fuzz languages, the focus and techniques of the two papers are completely different.

Groce et al.'s "swarm testing" [27] describes a program generation strategy that intentionally restricts itself to a subset of language features in order to focus testing on that chosen subset. For example, in a language with conditionals, loops, and assignment, one may choose to generate programs containing only loops and assignment in order to allow more in-depth testing of those features and their interactions with each other. While this technique theoretically gives up on finding certain bugs (i.e., those that arise from the eliminated language features), in practice it has been shown that swarm testing ends up finding more bugs in a given timeframe than does testing on the entire language at once. The reason why is that generating programs which hammer on a particular language feature or interaction between specific features becomes much more likely in this restricted space, and bugs tend to involve only a handful of features. Groce et al. have shown that this technique works for fuzzing languages based on syntactic features and for testing APIs. We extend this idea with our CLP-based technique by developing specialized fuzzers which individually target subsets of Rust's type system, as described in Section V-B.

Equivalence Modulo Inputs (EMI) is a compiler optimization and code generation testing technique wherein programs are generated which should behave in semantically identical ways given identical inputs [28]. That work explores inserting dead code into existing programs in order to generate "input-equivalent" programs. In our work, we employ code translations that take a given program and derive other programs which are "type-equivalent" (e.g., if the given program was well/ill-typed, then the derived programs will also be well/ill-typed for similar reasons). While the original EMI work is focused on testing optimizations and code generation, our work is specific to finding consistency bugs in typecheckers.

III. Generating Well-Typed Programs

In this section we describe how to phrase the problem of well-typed program generation in terms of constraint logic programming (CLP). We use as an example the problem of generating well-typed programs in System F [29], the polymorphically-typed lambda calculus. We use System F as a relatively simple way to explain the general ideas behind our approach. CLP is capable of expressing much more complicated type systems as demonstrated by our application of these ideas to Rust (Section V). No existing fuzzers can handle even something as simple as System F because of its higher-order functions and parametric polymorphism.

While CLP is a better solution for generating well-typed programs for language fuzzing than any current method, it is not a perfect solution. We conclude this section by describing some of the pitfalls of CLP with respect to well-typed program generation.

A. CLP for Program Generation

A well-known result in programming language theory states that logical propositions correspond to types and programs correspond to proof terms. This relation is often called the Curry-Howard Correspondence, after the logicians who first observed it. Given a logical proposition A, we can use the rules of logic to create a proof term M that encodes the proof of A. The Curry-Howard Correspondence states that A can be viewed as a type and M as a program of that type, written $M : A$. Thus, we can use a logical theorem prover to derive programs of a given type.

The space of possible provers we could use is vast. We observe that typing rules are written as nondeterministic inductive inference rules that operate over equality constraints. This observation naturally leads to the use of Prolog-like [30], [31] languages, which explicitly feature nondeterminism as well as equality constraints (via unification). We can then easily reuse existing tools (e.g., [7], [8]) because with Prolog-like languages, the execution model has a very close correspondence to common type system representations.

Throughout this paper we refer to Prolog-like languages as CLP languages, where CLP generalizes logic programming to integrate arithmetic constraint solvers [5], [6] (in fact, modern Prolog implementations such as GNU Prolog [8] and SWI-Prolog [7] are CLP languages in this sense). While typical type systems do not need arithmetic constraints, they are useful for more advanced type systems such as Rust's [10], as detailed in Section V-B.

B. Example: System F

The best way to explain how to use CLP for well-typed program generation is by example. Here we demonstrate how to use CLP to generate well-typed programs in System F, the polymorphically-typed lambda calculus. The key points to observe are that (1) the CLP specification closely mirrors the formal type system definition, and (2) we use only the standard features of CLP languages, which means that we can use off-the-shelf CLP implementations to generate programs.

Figure 1 describes the syntax of System F. Types are either type variables, function types, or polymorphic types.

$$\tau \in \mathit{Type} ::= \alpha \mid \tau_1 \to \tau_2 \mid \forall\alpha.\tau$$
$$e \in \mathit{Exp} ::= x \mid \lambda x{:}\tau\,.\,e \mid e_1\ e_2 \mid \Lambda\alpha\,.\,e \mid e\ \tau$$

Fig. 1: Syntax for System F, where α is a type variable, $\forall\alpha.\tau$ is a polymorphic type, x is a program variable, $\Lambda\alpha\,.\,e$ is a type abstraction that creates an expression of polymorphic type, and $e\ \tau$ instantiates a polymorphic expression to a specific type τ.

$$\frac{}{\Gamma, x:\tau \vdash \boldsymbol{x} : \tau}\ \text{VAR}$$

$$\frac{\Gamma, x:\tau_1 \vdash e : \tau_2}{\Gamma \vdash \boldsymbol{\lambda x{:}\tau_1\,.\,e} : \tau_1 \to \tau_2}\ \text{ABS}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \boldsymbol{e_1\ e_2} : \tau_2}\ \text{APP}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \boldsymbol{\Lambda\alpha\,.\,e} : \forall\alpha.\tau}\ \text{TABS}$$

$$\frac{\Gamma \vdash e : \forall\alpha.\tau_2}{\Gamma \vdash \boldsymbol{e\ \tau_1} : \tau_2[\alpha \mapsto \tau_1]}\ \text{TAPP}$$

Fig. 2: Typing rules for System F, where Γ is a type environment mapping variables to types and $\tau_2[\alpha \mapsto \tau_1]$ substitutes the type τ_1 for free occcurances of α in type τ_2.

An expression is either a variable, a function abstraction, a function application, a type abstraction, or a type application. Type abstractions parameterize an expression by a type in the same way that function abstractions parameterize an expression by a value. Type application specializes an expression (the body of a type abstraction) to a given type in the same way that function application specializes an expression (the body of a function abstraction) to a given value.

Figure 2 describes the typing rules for System F. The rules use *judgements* of the form "$\Gamma \vdash e : \tau$". A judgement is a statement "given type environment Γ that maps the variables in scope to their types, the expression e has type τ". Each rule has a single judgement as its conclusion (on the bottom of the horizontal line) and zero or more judgements as premises (on the top of the horizontal line). A rule provides justification for drawing the given conclusion only if we can prove the truth of each premise. The first three rules (VAR, ABS, and APP) are exactly the same as the simply-typed lambda calculus. The last two rules handle polymorphism: TABS introduces a polymorphic type and TAPP eliminates a polymorphic type.

Figure 3 shows a translation of the formal typing rules from Figure 2 into CLP, using a Prolog-like syntax where lower-case identifiers represent user-defined predicates and functions, capitalized identifiers represent variables, commas represent conjunction, `:-` represents reverse implication, and periods signal the end of a clause. All variables are implicitly quantified, e.g., the clause `foo(A,B) :- bar(A,C)` actually represents the clause `∀A,B.foo(A,B) :- ∃C.bar(A,C)`, which means "given some `A` and `B`, we can derive `foo(A,B)` if we can show there is some `C` such that `bar(A,C)`".

```
typing(Gamma,var(X),T) :-
    lookup(Gamma,X,T).
typing(Gamma,lam(X,T1,E),arrow(T1,T2)) :-
    add(X,T1,Gamma,NewGamma),
    typing(NewGamma,E,T2).
typing(Gamma,app(E1,E2),T2) :-
    typing(Gamma,E1,arrow(T1,T2)),
    typing(Gamma,E2,T1).
typing(Gamma,tlam(A,E),poly(A,T)) :-
    typing(Gamma,E,T).
typing(Gamma,tapp(E,T1),T3) :-
    typing(Gamma,E,poly(A,T2)),
    subst(A,T1,T2,T3).

?- typing([],E,T), write(E), fail.
```

Fig. 3: CLP specification of System F, where **lookup**(), **add**(), and **subst**() are helper predicates with the obvious functionality whose definitions are not shown here. The final query will output an infinite stream of well-typed System F programs.

The figure contains five clauses (one for each typing rule in Figure 2) and a query, which represents what the CLP engine should try to prove. The **`typing`** predicate represents a type judgement: **`typing(`**`Gamma,E,T`**`)`** stands for $\Gamma \vdash e : \tau$. The head of a clause (the part before the `:-` symbol) is the conclusion of the inference rule; the body of a clause contains the premises of the inference rule. The type environment Γ is represented using a list associating variables with their types; we use the helper predicate **`lookup(`**`Gamma,X,T`**`)`** to determine what type `T` is associated with variable `X` in type environment `Gamma`, and the helper predicate **`add(`**`X,T,Gamma,NewGamma`**`)`** to compute a new type environment `NewGamma` copied from the original type environment `Gamma` but associating variable `X` with type `T`. We also use the helper predicate **`subst(`**`A,T1,T2,T3`**`)`** to compute a new type `T3` derived from `T2` but with all free instances of type variable `A` replaced with type `T1`, i.e., `T3` $=$ `T2`$[$`A` $\mapsto$ `T1`$]$. We omit the definitions of these helper predicates from the figure for space reasons, but the entire code listing is available in the supplementary materials[1].

Consider the query at the bottom of Figure 3. Because variables are implicitly quantified, it actually represents the query "`∃E,T.typing([],E,T), write(E), fail`". In other words, "prove there exists some expression `E` and type `T` such that given the empty type environment [] expression `E` has type `T`, then output the expression `E`, then fail". In order to satisfy the first conjunct, the CLP engine must construct a satisfying expression and its type; the second conjunct is a built-in side-effecting operation that outputs its argument; the third conjunct will immediately fail. Because CLP is nondeterministic, failure triggers backtracking—the engine will backtrack to the last nondeterministic decision it made and make a different one. This implies that the engine will find a *different* expression `E` with some type `T` (assuming each expression has at most one type), output it, then fail again. This process will continue indefinitely, attempting to output the infinite set of well-typed System F programs.

There is, however, a subtle problem with how programs in System F are enumerated with this example. When performing nondeterministic search over clauses, CLP engines employ a

[1] http://www.cs.ucsb.edu/~kyledewey/ase15.zip

depth-first search strategy, executing alternatives in the order they are presented. For this example, this means the CLP engine will always choose to produce ever deeper lambda abstractions, as opposed to employing some of the later rules. This problem can be easily addressed by adding code to make lambda abstraction and other alternatives fail under certain conditions, causing the CLP engine to backtrack and choose a different alternative. Different mechanisms of causing failure correspond to different search strategies; for example, adding a bound on the number of recursive calls made to `typing` implements a bounded-exhaustive search, and adding probabilistic failure amounts to performing a random search. More details on alternate search strategies and their implementation can be found in our prior work. [26]

C. Pitfalls of CLP

While there are substantial advantages to using CLP for well-typed program generation, it does have certain problems that make it an imperfect solution. We identify and describe the two biggest problems we have encountered when using CLP for this purpose.

Fundamental Performance Issues. Typing rules generally assume that they are operating over complete programs and are attempting to make a judgement whether that program is well-typed, i.e., they are operating as *acceptors* rather than *generators*. Ideally, when implemented in CLP any acceptor is also a generator by default—given a predicate p that describes terms with some desired property, $p(t)$ operates as an acceptor for a concrete term t while $\exists x.p(x)$ operates as a generator that will bind x to some satisfying concrete term. However, naively translating typing rules into CLP can lead to performance issues. For example, while the ordering of clauses in a conjunction is irrelevant from a strictly logical standpoint, in practice it is significant. A poor ordering can lead to asymptotically worse performance [32], or even nontermination [33]. We have also found that it may be necessary to place bounds on types to ensure termination, using techniques described at the end of Section III-B.

Lack of Constructive Negation. CLP in general lacks the ability to constructively negate a predicate. In other words, it is not possible to have the CLP engine construct a term that deliberately *fails* to satisfy a given predicate. Given a predicate p, we can query $\exists x.p(x)$ but we cannot query $\exists x.\neg p(x)$. CLP languages often implement a notion called "negation by failure", but that is not constructive, i.e., it cannot construct terms, only filter out unsatisfying terms [34], [35]. In order to get the effect of constructively negating a predicate p, we must create a new predicate $\bar{p}$ that constructively describes the negation of p. This new predicate will contain redundant code, and the resulting specifications are longer and more confusing. Attempts have been made to solve this problem (e.g., [36]), but those solutions require specialized CLP implementations and still require additional code and effort.

IV. Finding Typechecker Bugs

A language's type system provides guarantees about program behavior, i.e., it excludes behaviors that the language developers have deemed "bad". The type system is essentially a logical theory by which the typechecker attempts to prove that a program does not exhibit these bad behaviors. One can think of the typechecker as a filter which allows through all programs that it can guarantee are well-behaved, while forbidding all programs that it cannot guarantee are well-behaved.

Because exactly determining which programs are well-behaved or ill-behaved is provably undecidable, the typechecker will conservatively reject some potentially well-behaved programs; the fewer such programs it rejects, the more *precise* the typechecker is. However, the typechecker should never accept any program that is potentially ill-behaved; this requirement is called *soundness*. In addition, the typechecker should be *consistent* in its decisions to avoid programmer confusion: similar programs from a typing perspective should all be accepted or rejected similarly.

In this section we describe methods and techniques for detecting bugs in a typechecker implementation. We focus specifically on **precision bugs**, **soundness bugs**, and **consistency bugs**, though at the end we also discuss a few other kinds of bugs that we encounter as a side-effect of our main focus. Determining what exactly constitutes a bug requires a specification to compare against. A formal specification would be best (for typecheckers, this would be a formal type system) but is not always available, especially for a language under rapid development. In the absence of a formal specification, we rely on an informal notion of "developer intent", gleaned from discussions with the language developers themselves. When we say "spec" below, we are referring to either the formal or informal specification, whichever is available. We do not consider the problem of determining whether the spec itself is correct (e.g., proving the soundness of the type system), though that is an interesting problem to tackle in the future.

A. Finding Precision Bugs

We wish to automatically generate programs that expose precision bugs in the typechecker. We first need a definition that tells us when a program exposes a precision bug:

Definition 1. ⟨***Precision Bug***⟩ *A program exposes a typechecker precision bug if the program is well-typed according to the spec but the typechecker rejects the program.*

Because the spec can be informally defined, it may be uncertain whether the program is well-typed according to the spec. Even if we can guarantee that the program is well-behaved, the type system implemented by the typechecker may not be able to prove that fact and in that case the program *should* be rejected by the typechecker. However, we can give a more specific condition under which the program should probably have been accepted:

Corollary. *A program exposes a precision bug if the typechecker has computed information that implies the program is well-typed, but rejects the program anyway.*

For example, suppose that the typechecker can infer the program is well-typed if it can prove some proposition q. It has already proved proposition p, and it knows that $p \supset q$. Thus, the typechecker should be able to derive q and declare the program well-typed. However, if it ignores that information and rejects the program then we say it has a precision bug.

```
typing(Gamma,app(E1,E2),T2) :-
    typing(Gamma,E1,arrow(T1,T2)),
    typing(Gamma,E2,T3),
    \+ (T3 == T1).
```

Fig. 4: CLP specification of "almost well-typed" System F. The only change is to the clause for the APP rule, the rest of the clauses are the same as Figure 3 and are omitted here. The \+ operator is negation-by-failure.

From these definitions, it suffices to generate well-typed programs using the technique described in Section III, run them through the typechecker, and see whether the typechecker accepts them or not. If a program is rejected, then given a formal spec we are guaranteed that we have exposed a bug. Given an informal spec, we have exposed a case where either there is a bug in the typechecker or the language developers need to tweak their notion of well-typedness to refine the informal spec.

B. Finding Soundness Bugs

We wish to automatically generate programs that expose soundness bugs in the typechecker. We first need a definition that tells us when a program exposes a soundness bug:

Definition 2. ⟨***Soundness Bug***⟩ *A program exposes a typechecker soundness bug if the program is not well-typed according to the spec but the typechecker accepts it as valid.*

Thus, in order to expose soundness bugs we must generate ill-typed programs. In concept, this is trivial: simply generate syntactically valid programs and filter out all those that are well-typed (as the generated programs grow larger, the odds of a syntactically well-formed program also being well-typed tend to shrink exponentially). However, the resulting ill-typed programs are generally *obviously* ill-typed, such that even a buggy typechecker would probably be able to correctly reject them. Intuitively, we want the ill-typed programs to be *non-obvious* so that even a mostly-correct typechecker might still trip up and incorrectly accept them.

For this purpose, we introduce the notion of "almost well-typed" programs. The idea is simple: given a set of type system rules, we pick a subset of the rules' premises and negate them. Any program that is well-typed according to the modified type system is "almost well-typed" according to the original type system—that is, the program is ill-typed, but in a precisely controlled way. This notion is independent of the particular type system that the typechecker implements, allows us to tune the degree of ill-typedness at a fine granularity (by choosing how many and which premises to negate), and is intended to mirror likely mistakes that might be made when implementing the typechecker (for example, forgetting to check a rule's premise or checking it incorrectly).

Example: Almost Well-Typed System F. We illustrate this idea using the System F example from Section III. Consider Figure 2, which gives the typing rules for System F. Suppose that we decide to negate the second premise of the APP rule, which says $\Gamma \vdash e_2 : \tau_1$ (i.e., that the type of the argument matches the type of the function's parameter). Negating this premise means ensuring that the type of the argument e_2 is *not* the type of the function parameter τ_1. Figure 4 gives a modified implementation of the APP rule using CLP, which can be compared to the CLP implementation given in Figure 3. The clause in Figure 4 is generating a type `T3` for `E2` and then ensuring that `T3` is *not* the same as `T1`. We would prefer to use constructive negation to create a type `T3` that is different from `T1` by construction, however as discussed in Section III-C this is not possible in typical CLP languages. Thus, the almost well-typed implementation must spell out to the CLP engine what negation means in the context of each negated premise.

While this example requires that *every* function application in the generated program is ill-typed, we can also specify that only a certain *number* of function applications are ill-typed, e.g., that there is exactly one ill-typed function application and all the rest are well-typed. In general, for any "almost well-typed" program generation we can specify how many times each negated premise is used versus the original premise. We accomplish this by adding a counter that counts the number of times a negated premise is applied; if the counter exceeds some bound then the negated premise cannot be used anymore and the generator must use the original, non-negated premise.

C. Finding Consistency Bugs

We wish to automatically generate programs that expose consistency bugs in the typechecker. We first need a definition that tells us when two programs expose a consistency bug:

Definition 3. ⟨***Consistency Bug***⟩ *Two programs together expose a typechecker consistency bug if (1) the well-typedness (ill-typedness) of one implies the well-typedness (ill-typedness) of the other; and (2) the typechecker accepts one program and rejects the other.*

To find consistency bugs according to this definition, we need a method to generate "type-equivalent" programs—that is, programs that satisfy point (1) in Definition 3. Generally, type-equivalence is specific to the language under test. For expository reasons, we provide a simple example to illustrate the general idea. Consider the following program:

let x:τ = e_1 **in** e_2

Where τ is some type and e_1 and e_2 are arbitrary expressions. The overall meaning of this program is to evaluate e_1 down to a value of type τ, assign the result to the variable `x`, and then evaluate e_2 with `x` in scope. With this simple setup, we can automatically transform this program into the equivalent one below:

let t:τ = e_1 **in** (**let** x:τ = t **in** e_2)

Which preserves the meaning and typedness (whether well-typed or ill-typed) of the original program. If one of the above programs is accepted by a typechecker and the other is rejected, then the typechecker has a consistency bug.

In order to systematically check for consistency bugs, we first generate well-typed and almost well-typed programs according to the methods described in Sections IV-A and IV-B, then apply a series of language-specific transformations on the resulting programs to create type-equivalent sets of programs, then run each type equivalence class of programs through the typechecker to see whether they are all accepted or rejected.

D. Other Kinds of Bugs

While our main focus is on precision, soundness, and consistency bugs, in the process of finding them we can encounter other kinds of bugs as well. Two common kinds of bugs that we may encounter are **parser bugs** and **crash bugs**, as defined below.

Definition 4. ⟨***Parser Bug***⟩ *A program exposes a parser bug if it is syntactically well-formed according to the spec but the parser rejects it.*

Definition 5. ⟨***Crash Bug***⟩ *A program exposes a crash bug if it causes the compiler to crash when compiling it.*

Not all bugs neatly fit into the aforementioned categories. We call these **miscellaneous bugs**, as defined below:

Definition 6. ⟨***Miscellaneous Bug***⟩ *A program exposes a bug which is not clearly identifiable as a* ***precision****,* ***soundness****,* ***consistency****,* ***parser****, or* ***crash*** *bug.*

We do not do anything special to find these three additional kinds of bugs, but merely make a note when our testing encounters them.

V. Testing the Rust Typechecker

In this section we describe Mozilla's Rust language [10], with particular attention to its type system, as well as a set of program generators that we have implemented for the Rust typechecker using the techniques described in Section IV. Rust serves as an interesting case study for our techniques because it is under active development and features a sophisticated but informally-defined type system. The lack of formal specification means that we cannot establish ground truth regarding typedness, but must instead establish a dialogue with the Rust developers to evaluate the results of our testing. This situation is common in large-scale, industrial-strength language development, and our successful application of these techniques to Rust demonstrates that they can handle such languages. This work is the first to successfully generate well-typed Rust programs and to systematically test the Rust typechecker. All of the program generators described here are available in the supplementary materials[2].

A. Rust Background

Rust is intended to be a systems-level programming language along the lines of C and C++, but with much greater safety guarantees afforded by its type system. Rust supports tuples, records, generics, parametric polymorphism, type classes, associated types, linear types, and borrowing. We briefly describe some of the less common typing features: type classes, associated types, linear types, and borrowing.

Type Classes. First introduced in Haskell, type classes [37] provide a more principled way of allowing for ad-hoc type polymorphism. A type class declares a set of polymorphic function signatures that must be implemented by all members of that class. Polymorphic type variables can then be constrained to require that they belong to a given type class. Type classes are interesting from a well-typed program generation standpoint because determining well-typedness requires reasoning about type constraints arising from an intricate mixture of syntactic and semantic features.

Associated Types. A useful feature seen in Standard ML, C++, Haskell, and Rust, among others, is that of associated types [38], [39], [40], which are intended to simplify polymorphic code. This feature allows auxiliary type variables to be associated with some type τ, such that these auxiliary variables are *implicitly* passed whenever τ is *explicitly* passed. In practice, this feature can dramatically cut down on the number of type variables which must explicitly be passed in the code, greatly reducing boilerplate.

Linear Types. One of the most recognized features of Rust is its use of linear types [41], [42] over memory regions [43]. Rust did not pioneer the use of linear types (see, e.g., [44], [45] among others), but it is the first language to use them that has substantial industry support. Rust uses linear types for automated memory management without garbage collection or reference counting. By default, all variables are linearly typed. The key property that linear types enforce is that any linearly-typed variable is used exactly once. Intuitively, a linearly-typed variable's value is a resource that is consumed when that variable is used. If a linearly-typed variable goes out of scope and its associated value has not been consumed, then the underlying memory for that value can safely be reclaimed. Consider the following *ill-typed* Rust code:

```
fn dup1<A,B>(a:A, b:B) -> (A,A) { (a,a) }
```

This code declares a polymorphic function `dup1` with two parameters `a` and `b` whose return type is a tuple with elements the same type as parameter `a`. This code is ill-typed because `a` is used twice in the body of the function to construct the pair being returned. The following version is well-typed:

```
fn dup2<A,B>(a1:A, a2:A, b:B) -> (A,A) {
  (a1,a2)
}
```

The values of parameters `a1` and `a2` are consumed to produce the return value, while parameter `b` is unused and thus its value is unconsumed. Therefore, `b`'s value will be automatically reclaimed when `dup2` returns.

Borrowing. Linear typing is severely restrictive in practice, as shown in the previous `dup1` example where it was not possible to duplicate the parameter `a`. To alleviate this problem, Rust relaxes linearity in a sound manner using references. Intuitively, when the programmer creates a reference to a value then that reference *borrows* the value for a clearly-defined duration without consuming it. The borrow durations are made explicit in the type system via *lifetime variables*, which are associated with every reference and constrain how long a reference is permitted to borrow a value. Consider the following well-typed Rust program:

```
fn dupref<'a,A>(r: &'a A) -> (&'a A, &'a A) {
  (r, r)
}

fn calldup<A>(a: A) {
  let r = &a;
  let (d1, d2) = dupref(r);
}
```

[2]http://www.cs.ucsb.edu/~kyledewey/ase15.zip

Where `'a` is a lifetime variable. The `dupref` parameter `r` is a reference to a type `A` with lifetime `'a`. Unlike the data referenced, the reference itself can be treated nonlinearly, as shown by the return value which uses `r` twice to construct a tuple. Lifetime variables are created automatically by the compiler, as shown in `calldup` which creates the initial reference to `a`. The typechecker is responsible for verifying that reference lifetimes are properly observed to avoid memory safety violations.

B. Rust Program Generators

Rather than create a single program generator that attempts to encompass all of Rust at once, we create separate generators which focus in on different parts of Rust's type system, much in the same spirit as that of Groce et al. [27]. These generators are overall much more complex than the example shown in Section III-B, with each spanning several hundred lines of code. As such, due to space constraints, we generally can only describe from a high level what these generators do. We encourage readers to consult our supplementary materials[3], which contain complete code for all the generators.

Generator G1. This generator creates programs with well-typed first-order functions and function bodies. This generator handles memory regions, lifetime variables, first-order function calls, loops, variables, conditionals, and references. Because lifetime variables model the duration under which a reference is valid, we treat them as symbolic integers. This representation is amenable to CLP's built-in arithmetic constraint solvers. Overall, this feature set represents the very heart of Rust and typical programs would use this portion most frequently. This generator demonstrates CLP's capability to handle the core features of Rust's type system. Unfortunately, G1 is occasionally less precise than Rust's typechecker in ways which are difficult to address without augmenting G1 with a dataflow analysis. As such, we did not experiment with generating nearly well-typed programs from G1, as such programs may still be well-typed in Rust thanks to additional information G1 does not track.

Generator $G2_W$. This generator creates programs with well-typed records, type classes, and type class implementations. This generator internally implements a simple, specialized constraint solver over type constraints, which mirrors a similar constraint solver in Rust's typechecker implementation [46]. A sanitized snippet from our constraint solver implementation is shown in Figure 5, which has been stripped of code related to bounded-exhaustive search and has undergone some variable and procedure renaming. Overall, CLP is highly amenable to this approach of implementing custom constraint solvers, a fact which has been noted elsewhere [47].

Generator $G2_I$. We used our technique for "almost well-typed" program generation (see Section IV-B) and modified the $G2_W$ generator to create ill-typed programs for the same subset of Rust. This was achieved by nondeterministically skipping calls to the **`constraintHolds`** procedure shown in Figure 5, leading to the introduction of constraints with unverified validity. Because a constraint may still hold by chance, an additional check was performed at the end of generation to ensure that the program was not accidentally well-typed, much like the check performed in Figure 4.

[3] http://www.cs.ucsb.edu/~kyledewey/ase15.zip

```
constraintHolds(
   State1, LifetimeCons, FinalState) :-
  LifetimeCons =
    lifetimeCons(Lifetime1, Lifetime2),
  lifetimeInScope(State1, Lifetime1),
  lifetimeInScope(State1, Lifetime2),
  ensureLivesAtLeast(Lifetime1, Lifetime2),
  addAssumption(
    State1, LifetimeCons, FinalState).

constraintHolds(
   State1, TypeLifetimeCons, FinalState) :-
  TypeLifetimeCons =
    typeLifetimeCons(Type, Lifetime),
  lifetimeInScope(State1, Lifetime),
  inhabitedType(State1, Type, State2),
  addAssumption(
    State2, TypeLifetimeCons, State3),
  handleImpliedLifetimeConstraints(
    State3, Type, Lifetime, FinalState).
```

Fig. 5: Snippet of sanitized code handling two of the three possible Rust type constraints we consider, which is used as a constraint solver. The first case of `constraintHolds` handles the Rust constraint `'lt1 : 'lt2`, meaning the lifetime `'lt1` lives at least as long as the lifetime `'lt2`. The second case handles the constraint `Type : 'lt`, meaning the type `Type` lives at least as long as the lifetime `'lt`. Descriptions of select data involved and called procedures is provided inline below.

`State` Holds everything in scope, including lifetime variables, type variables, typeclasses, typeclass implementations, and previous typing assumptions made. Different actions manipulate the state, resulting in new states.

`lifetimeInScope(S,L)` succeeds if the lifetime variable `L` is in scope with respect to state `S`.

`ensureLivesAtLeast(L1,L2)` succeeds if lifetime `L1` lives at least as long as `L2`, potentially adding a CLP arithmetic constraint roughly of the form $L1 \leq L2$, where smaller values represent longer-living lifetimes.

`addAssumption(S1,C,S2)` records that the constraint `C` has been assumed to be true. `C` is added to state `S1` to yield state `S2`.

`inhabitedType(S1,T,S2)` succeeds if the type `T` is inhabited under state `S1`, yielding state `S2`. Depending on `T`, `constraintHolds` may end up being called in a mutually recursive fashion, as determining if an arbitrary type is inhabited in Rust entails checking constraints on types. New information can result from these recursive calls, hence the need for state `S2`.

`handleImpliedLifetimeConstraints(S1,T,L,S2)` records any information implied by the fact that the type `T` has been shown to live at least as long as lifetime `L`. For example, if `T` is `&'a Foo` and `L` is `'b`, then we also know that `Foo : 'a` and `'a : 'b`. New information is added to state `S1`, yielding state `S2`.

Generator G3. This generator also creates programs with records, type classes, and type class implementations, but in a much less constrained way than generator G2. It handles a larger subset of Rust, including associated types, but does not guarantee that the generated programs are well-typed and neither does it guarantee that they are "almost well-typed"; instead, we use the techniques described in Section IV-C to create type-equivalence classes of programs. The purpose of this generator is to find consistency bugs, and so it doesn't matter whether the programs are well-typed or not—only that the typechecker treats them consistently. Our program transformations to create type-equivalent programs mainly move the

placement of explicit type constraints in the generated code such that the movement should have no effect on typedness—the concept is analogous to replacing $A \wedge B$ with $B \wedge A$. While there is overlap between this generator and others, G3 is specifically targeted to find consistency bugs with high likelihood.

Generator $\mathbf{G4_W}$. This generator is similar to $G2_W$ but the internal constraint solver is even more precise, enabling the generator to recognize more programs as being well-typed. This internal constraint solver takes more implied type information into account than in $G2_W$, and is generally even more precise than the constraint solver implemented in the Rust typechecker. This serves to point out places where the Rust typechecker needlessly loses precision. This is actually a defective early prototype of $G2_W$, and bugs found by $G4_W$ were used to help inform how Rust's typechecker works. Because the behavior of $G4_W$ intentionally differs from Rust, it quickly finds many possibly duplicate bugs, necessitating triage. As a result, there are potentially more issues to be found with $G4_W$ than what we report.

Generator $\mathbf{G4_I}$. We used our technique for "almost well-typed" program generation (see Section IV-B) and modified the $G4_W$ generator to create ill-typed programs for the same subset of Rust. This was done in the exact same way as with the creation of $G2_I$, by nondeterministically skipping calls to the `constraintHolds` procedure. As with $G4_W$, this quickly finds many issues, and so this generator may have actually found more distinct issues than what we report.

VI. Evaluation

We evaluate our techniques for typechecker fuzzing by implementing the program generators described in Section V and applying them to test the Rust language implementation. Through this process we have uncovered 18 bugs according to the definitions given in Section IV: 12 that have been acknowledged by the Rust developers, 2 that the developers are still considering, and 4 that the developers do not wish to consider as bugs. We report the issues we uncovered that the developers have decided not to treat as bugs because (1) they are still bugs according to our definitions; and (2) uncovering these issues often led to in-depth discussion and debate by the developers before deciding that they were not bugs—as such, the questions raised by these issues were useful for tweaking the informal spec even if they did not result in fixes to the language.

A. Experimental Details

Rust is under active development, thus for our experiments we fixed on testing version 1.0-alpha, which was the most recent version circa the start date of this work. In order to carry out our testing, we implemented a custom SMP parallel fuzzing tool written in a combination of Scala [48] and Rust itself. The Rust portion takes advantage of the fact that Rust compiler internals are exported as a library, allowing us to repeatedly call specifically into the Rust typechecker without incurring the overhead of repeatedly loading in the Rust compiler as a process. Across the entire tool, disk IO occurs only to write out newly discovered bugs; all other interprocess communication occurs through UNIX pipes. Our testing infrastructure and the program generators that we evaluated are all available in the supplementary materials[4].

We dynamically ensure that duplicate crash bugs are unique using the techniques described in Chen et al. [49]. Automatically detecting duplicate typechecker bugs is an open problem; we did so manually for these experiments, and retroactively modify the generators to avoid repeatedly hitting the same bug where possible. Over a period of over 600 machine-hours we tested nearly 900 million generated programs. In terms of performance, the testing of generated programs turns out to be the bottleneck rather than program generation itself: a single program generation process was always able to outpace the testing of the resulting programs (often by orders of magnitude) even when the testing was carried out in parallel threads on a 36-core machine. For example, the $G2_W$ fuzzer generates approximately 138k programs per minute, though we can only test approximately 28k per minute on a 12 core machine. Moreover, the test harness has been heavily optimized, whereas the generators are fairly naive.

B. Reported Bugs

Table I lists the bugs that we uncovered during testing. For each bug we report which program generator was used to find it, its status (whether the developers have confirmed it as a bug, decided it is *not* a bug, or have yet to make an explicit determination), and give a brief description of the bug. We discuss the causes and implications of a select few of these bugs below.

Optimistic Treatment of Unsound Expressions. Rust allows for modular code definitions, which impacts typechecking of type classes. Specifically, when type class implementations are separate from type class declarations, there is an issue determining whether or not a given type class implementation properly implements its supposed type class. Concretely, suppose that the programmer adds a constraint that type `bool` (a built-in primitive type) must implement some user-defined type class `TC`. The actual implementation of `TC` for `bool` may not be present in the code being compiled (for example, if that code is a library which will be linked in to other code downstream). This fact raises an ambiguity with respect to the meaning of the constraint, which could be (1) `bool` must immediately implement `TC` in currently available code, or (2) `bool` must *eventually* implement `TC`, including in code that is not yet available. Experimentally using our tests we have determined that the Rust typechecker takes the latter approach, which is not consistent with the expected behavior from certain bug reports [60], and ultimately leads to bugs 9 and 11 in Table I. We label this issue as a soundness bug because the typechecker can optimistically accept unsound code in the vain hope that it will eventually get enough information to prove that the code is sound. The result is that, for example, library code can compile properly, but if a user ever links to that library *and* attempts to use the unsound code, the compiler will unexpectedly reject their code with a very opaque and useless error message. The underlying problem is currently being addressed [52], and ultimately a breaking language change will result.

[4] http://www.cs.ucsb.edu/~kyledewey/ase15.zip

TABLE I: Summary of reported issues and bugs. "Confirmed?" is ✓ (developers confirm as a bug), ✗ (developers decided it's *not* a bug). References to Rust language Github issues are provided where possible.

Bug ID	Kind	Generator	Confirmed?	Brief Description
1	Precision	G1	✗	The type system discards some information around blocks, leading to the rejection of well-behaved programs.
2	Precision	G1	✗	The type system conservatively considers it possible for two references which point to incompatible types to assign into each other.
3	Precision	G1	✗	Forcing lifetime variables to be equivalent triggers precision loss.
4	Precision	G1	✗	In general, it is not possible to refactor code so that an expression is replaced by a call to a function that performs the same operation as the original expression.
5	Precision	$G2_W$	✓	The compiler rejects a program saying that an additional, but irrelevant, constraint is needed.
6	Precision	$G2_W$	✓	Constraints of the form `typ : ...` behave in fundamentally different ways depending on whether or not `typ` is a type variable.
7	Precision	$G2_W$	✓ [50]	Programs which fail to use all lifetime and type variables available in certain syntactic positions are rejected, ultimately to make implementation simpler.
8	Precision	$G4_W$	✓ [51]	The compiler discards implied information in the context of typeclasses.
9	Soundness	$G2_I$	✓ [52]	The typechecker does not check that the type we implement a typeclass for can actually exist.
10	Soundness	$G2_I$	✓ [53]	Constraints over lifetime or type variables are discarded if they are unused in certain syntactic positions.
11	Soundness	$G4_I$	✓	The solving of type constraints on typeclass implementations which check that a given type implements a given typeclass are delayed until a typeclass is used.
12	Consistency	G3	✓ [54]	Syntactically duplicating a constraint on a type variable in the type variable position is considered a type error, when logically this is the same as saying $A \wedge A$
13	Consistency	G3	✓ [55]	If two constraints on a type variable in the type variable position refer to the same typeclass with different type parameters, the compiler incorrectly reports the are syntactically identical.
14	Parser	G1	✓ [56]	Fails to parse "`box ()`", which should heap-allocate the unit (`()`) value
15	Crash	$G2_W$	✓ [57]	The solving of constraints involving lifetime parameters are delayed until after they are needed, resulting in internal inconsistencies.
16	Crash	G3	✓ [58]	A crash occurs if a type constraint on a typeclass attempts to refer to its own associated type.
17	Miscellaneous	G1	✓ [59]	Untouched contents of a `struct` can impact typechecking.
18	Miscellaneous	G1	✓	Lifetime variables do not correspond directly to memory regions [43].

Lack of Modus Ponens. Because types are propositions, it is possible to use the existence of a type to infer further type constraints using the rules of logic. That is, if we know p and we know $p \supset q$ then we can logically infer q; this is the well-known rule of modus ponens. However, it turns out that Rust does not always apply this rule during typechecking, leading to imprecision and unintuitive behavior. For example, when checking whether a type class is properly implemented, the Rust typechecker inconsistently applies modus ponens when checking type constraints, leading to bugs 6 and 8 in Table I. The developers acknowledge that this problem is a fundamental issue, and there are plans to address it [51], [52].

Inconsistent Handling of Type Constraints. Rust allows explicit type constraints to be placed on various language features (e.g., a function definition). To improve flexibility and readability [61], the programmer can place these constraints in either of two syntactic positions. Logically, the syntactic placement of the constraint should have no bearing on its meaning—however, our testing revealed that constraints in different syntactic positions are handled inconsistently (see bugs 12 and 13 in Table I). The Rust developers have since fixed the underlying problem.

VII. Conclusion

We have proposed a general technique for using CLP to fuzz typechecker implementations of statically-typed languages with advanced type systems. We generate well-typed programs with this technique, which is a major accomplishment in and of itself; we also describe a set of novel techniques to target precision, soundness, and consistency bugs in the typechecker. We have applied this technique to fuzz the typechecker of Rust, a complex, real-world language. Through this process we have identified 18 bugs in the Rust typechecker.

Acknowledgements. We wish to thank Niko Matsakis and the Rust development team for their help in this project.

REFERENCES

[1] J. Ruderman, "Introducing jsfunfuzz," 2007. [Online]. Available: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/

[2] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 38–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362831

[3] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993532

[4] W. Howard, "The formulae-as-types notion of construction," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. Seldin and J. Hindley, Eds. Academic Press, 1980, pp. 479–490.

[5] J. Jaffar and J.-L. Lassez, "Constraint logic programming," in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ser. POPL '87. New York, NY, USA: ACM, 1987, pp. 111–119. [Online]. Available: http://doi.acm.org/10.1145/41625.41635

[6] J. Jaffar and M. J. Maher, "Constraint logic programming: A survey," *Journal of Logic Programming*, vol. 19, pp. 503–581, 1994.

[7] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[8] D. Diaz and P. Codognet, "The gnu prolog system and its implementation," in *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 2*, ser. SAC '00. New York, NY, USA: ACM, 2000, pp. 728–732. [Online]. Available: http://doi.acm.org/10.1145/338407.338553

[9] R. A. O'Keefe, *The Craft of Prolog*. Cambridge, MA, USA: MIT Press, 1990.

[10] Mozilla, "The rust language website." [Online]. Available: http://www.rust-lang.org/

[11] W. M. McKeeman, "Differential testing for software." *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, December 1998.

[12] M. Zalewski, "Announcing cross_fuzz: a potential 0-day in circulation, and more," 2011. [Online]. Available: http://lcamtuf.blogspot.com/2011/01/announcing-crossfuzz-potential-0-day-in.html

[13] J. Ruderman, "Jesse's fuzzer for arithmetic correctness in tracemonkey," 2008. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=465274

[14] V. St-Amour and N. Toronto, "Experience report: applying random testing to a base type environment," in *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '13. New York, NY, USA: ACM, 2013, pp. 351–356. [Online]. Available: http://doi.acm.org/10.1145/2500365.2500616

[15] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in udita," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 225–234. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806835

[16] C. Pacheco and M. D. Ernst, "Eclat: Automatic generation and classification of test inputs," in *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, Glasgow, Scotland, July 27–29, 2005, pp. 504–527.

[17] C. Csallner and Y. Smaragdakis, "Jcrasher: An automatic robustness tester for java," *Softw. Pract. Exper.*, vol. 34, no. 11, pp. 1025–1050, Sep. 2004. [Online]. Available: http://dx.doi.org/10.1002/spe.602

[18] B. Fetscher, K. Claessen, M. Palka, J. Hughes, and R. B. Findler, "Making random judgements: Automatically generating well-typed terms from the definition of a type-system," eSOP 2015.

[19] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler, "Run your research: On the effectiveness of lightweight mechanization," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 285–296. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103691

[20] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 281–294. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065045

[21] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter, "Synthesis modulo recursive functions," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 407–426. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509555

[22] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. Program. Lang. Syst.*, vol. 1, no. 2, pp. 245–257, Oct. 1979. [Online]. Available: http://doi.acm.org/10.1145/357073.357079

[23] D. Kroening and O. Strichman, *Decision Procedures: An Algorithmic Point of View*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008. [Online]. Available: http://books.google.com/books?id=anJsH3Dq5BIC

[24] B. C. Pierce, *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002.

[25] K. R. M. Leino, "Automating induction with an smt solver," in *Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 315–331. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27940-9_21

[26] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 725–730. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642963

[27] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 78–88. [Online]. Available: http://doi.acm.org/10.1145/04000800.2336763

[28] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 216–226. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594334

[29] J. C. Reynolds, "Towards a theory of type structure," in *Programming Symposium, Proceedings Colloque Sur La Programmation*. London, UK, UK: Springer-Verlag, 1974, pp. 408–423. [Online]. Available: http://dl.acm.org/citation.cfm?id=647323.721503

[30] P. Roussel, *Prolog Manual de Reference et d'Utilisation*. Groupe d'Intelligence Artificielle der Marseille Lumimy, 1975. [Online]. Available: http://books.google.com/books?id=Yj7qMgEACAAJ

[31] D. H. D. Warren, L. M. Pereira, and F. Pereira, "Prolog - the language and its implementation compared with lisp," *SIGART Bull.*, no. 64, pp. 109–115, Aug. 1977. [Online]. Available: http://doi.acm.org/10.1145/872736.806939

[32] U. S. Reddy, "A typed foundation for directional logic programming," in *In Proc. Workshop on Extensions to Logic Programming*, 1992, pp. 199–222.

[33] L. Sterling and E. Shapiro, *The Art of Prolog (2Nd Ed.): Advanced Programming Techniques*. Cambridge, MA, USA: MIT Press, 1994.

[34] K. L. Clark, "Readings in nonmonotonic reasoning," M. L. Ginsberg, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, ch. Negation As Failure, pp. 311–325. [Online]. Available: http://dl.acm.org/citation.cfm?id=42641.42664

[35] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1993.

[36] D. Pearce and G. Wagner, "Logic programming with strong negation," in *Extensions of Logic Programming*, ser. Lecture Notes in Computer Science, P. Schroeder-Heister, Ed. Springer Berlin

Heidelberg, 1991, vol. 475, pp. 311–326. [Online]. Available: http://dx.doi.org/10.1007/BFb0038700

[37] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '89. New York, NY, USA: ACM, 1989, pp. 60–76. [Online]. Available: http://doi.acm.org/10.1145/75277.75283

[38] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock, "A comparative study of language support for generic programming," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 115–134. [Online]. Available: http://doi.acm.org/10.1145/949305.949317

[39] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow, "Associated types with class," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: ACM, 2005, pp. 1–13. [Online]. Available: http://doi.acm.org/10.1145/1040305.1040306

[40] "Rust rfc for associated types and items." [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/0195-associated-items.md

[41] J. Girard, "Linear logic," *Theor. Comput. Sci.*, vol. 50, pp. 1–102, 1987. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(87)90045-4

[42] P. Wadler, "Linear types can change the world!" in *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[43] D. Gay and A. Aiken, "Memory management with explicit regions," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 313–323. [Online]. Available: http://doi.acm.org/10.1145/277650.277748

[44] K. Crary, D. Walker, and G. Morrisett, "Typed memory management in a calculus of capabilities," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99. New York, NY, USA: ACM, 1999, pp. 262–275. [Online]. Available: http://doi.acm.org/10.1145/292540.292564

[45] J. A. Tov and R. Pucella, "Practical affine types," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 447–458. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926436

[46] "Rust language constraint solver." [Online]. Available: https://doc.rust-lang.org/1.0.0-alpha/rustc_trans/middle/traits/struct.FulfillmentContext.html

[47] A. P. Felty, "Specifying and implementing theorem provers in a higher-order logic programming language," Tech. Rep., 1989.

[48] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*, 2nd ed. USA: Artima Incorporation, 2011.

[49] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 197–208. [Online]. Available: http://doi.acm.org/10.1145/2462156.2462173

[50] "Rfc: No unused impl parameters." [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/0447-no-unused-impl-parameters.md

[51] "Implied bounds: Rust does not always infer everything possible." [Online]. Available: http://smallcultfollowing.com/babysteps/blog/2014/07/06/implied-bounds/

[52] "New projections, lifetimes, and well-formedness rfc indicates check was previously not performed." [Online]. Available: https://github.com/nikomatsakis/rfcs/blob/projection-and-lifetimes/text/0000-projections-lifetimes-and-wf.md

[53] "Type bounds are conditionally enforced on impls." [Online]. Available: https://github.com/rust-lang/rust/issues/25110

[54] "Bug in checker for duplicate constraints." [Online]. Available: https://github.com/rust-lang/rust/issues/18693

[55] "Same trait with different input concrete parameters are considered duplicate bounds." [Online]. Available: https://github.com/rust-lang/rust/issues/22279

[56] "Box and in for stdlib: Ambiguity involving box and expressions surrounded by parentheses." [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/0809-box-and-in-for-stdlib.md

[57] "Coherence failed to report ambiguity ice involving lifetimes." [Online]. Available: https://github.com/rust-lang/rust/issues/24424

[58] "Ice using associated type in trait bound." [Online]. Available: https://github.com/rust-lang/rust/issues/20220

[59] "Rfc: Variance: The need for phantomdata." [Online]. Available: https://github.com/nikomatsakis/rfcs/blob/variance-redux/text/0000-variance.md

[60] "Type bounds are ignored in trait and impl function definitions." [Online]. Available: https://github.com/rust-lang/rust/issues/24011

[61] "Rfc: Where clauses." [Online]. Available: https://github.com/rust-lang/rfcs/blob/master/text/0135-where.md