Fuzzing the Rust Typechecker Using CLP

Kyle Dewey, Jared Roesch, Ben Hardekopf

University of California, Santa Barbara

Teaser

- We identify three kinds of bugs that typecheckers can exhibit
- We describe general techniques for automatically finding these kinds of bugs
- We apply these ideas to testing the Rust programming language, and find 14 developer-confirmed bugs

Outline

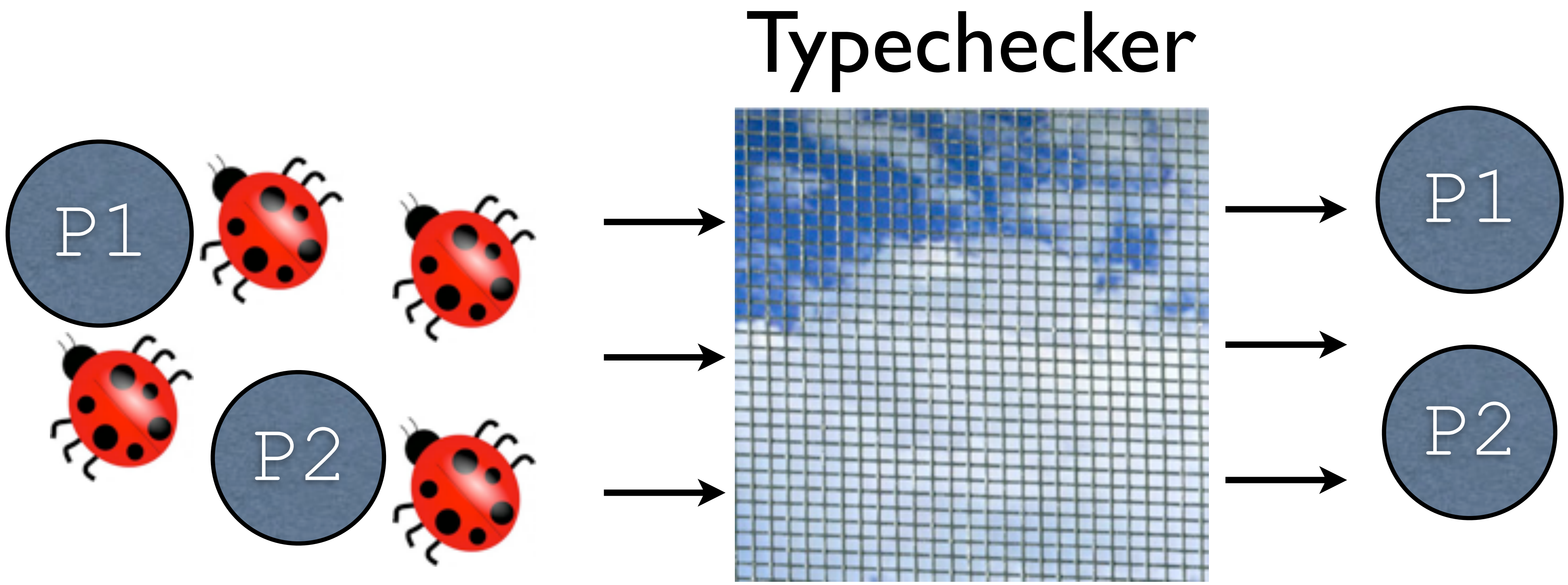
- Background and motivation
- Finding *precision* bugs
- Finding *soundness* bugs
- Finding *consistency* bugs
- Application to Rust
- Results

Outline

- Background and motivation
- Finding *precision* bugs
- Finding *soundness* bugs
- Finding *consistency* bugs
- Application to Rust
- Results

Motivation

- Typecheckers are crucial components in statically typed languages
  - Help ensure programs are correct
  - Defend against exploits



Typechecker

P1

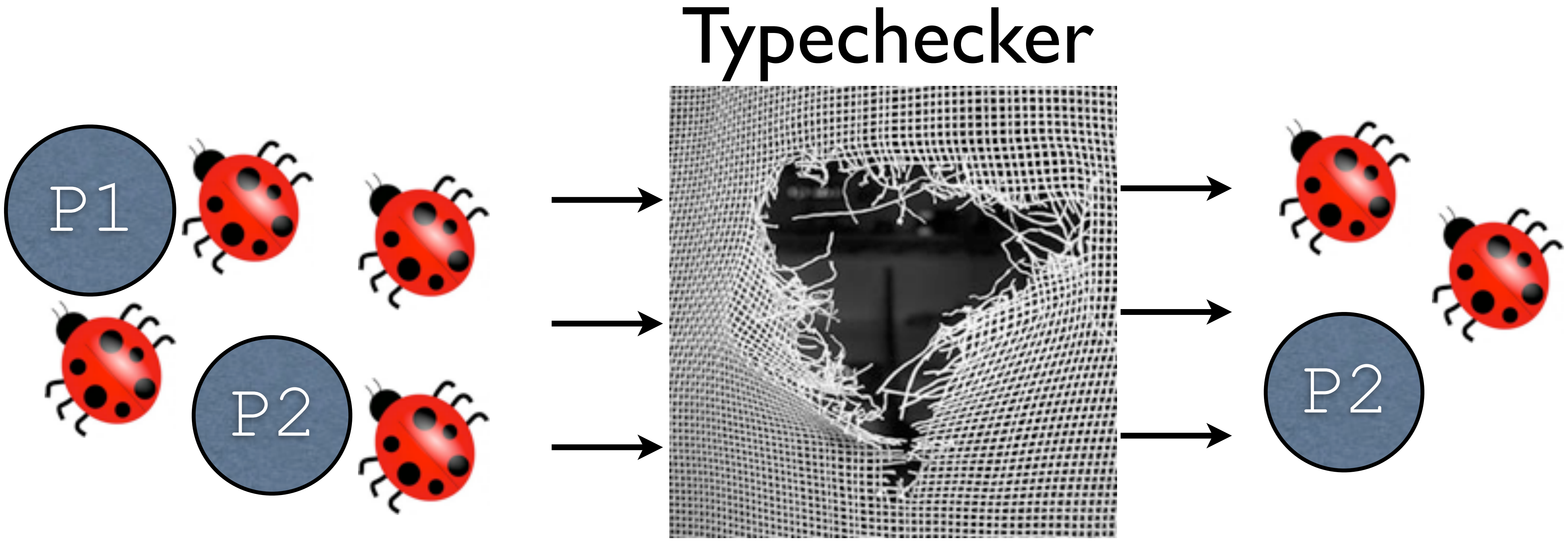
P2

P1

P2

Motivation

- Problem: typecheckers can be buggy too
    - Fail to accept well-typed programs
    - Fail to reject ill-typed programs

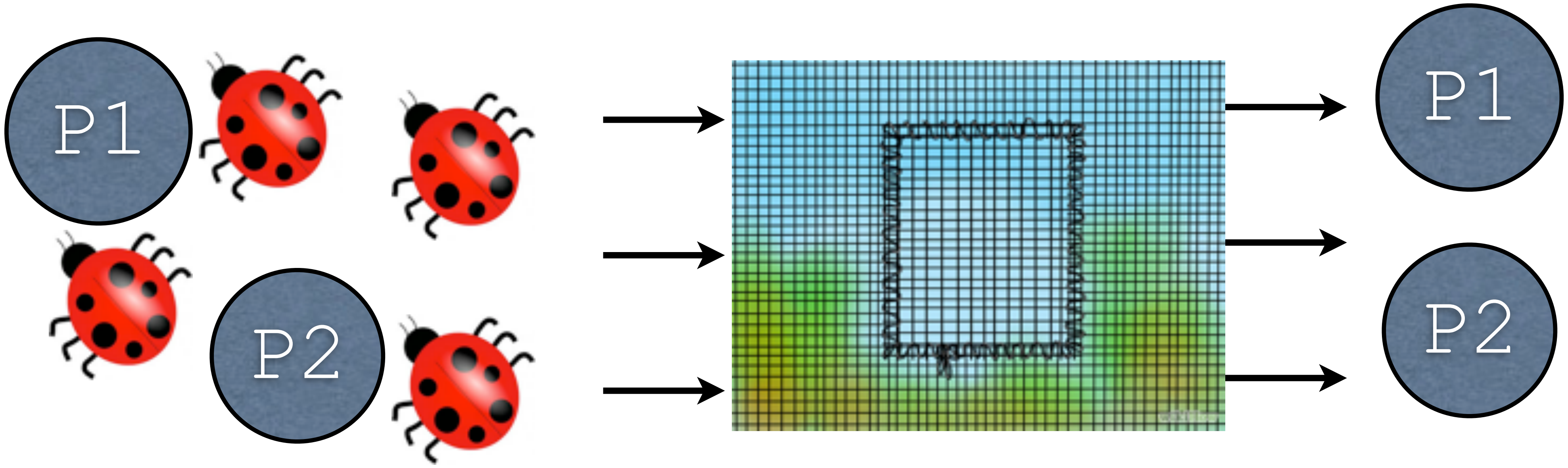


Typechecker

Idea

- Use black-box *language fuzzing* techniques to automatically find these bugs, ideally before they become a problem

---



Typechecker

Existing Work

- Most existing work on language fuzzing fundamentally applies only to dynamically-typed languages (e.g., `jsfunfuzz`)
- Based on performing a random walk over the language's grammar, referred to as a *stochastic grammar*

Stochastic Grammars

$$\oplus \in Binop ::= + \mid \&\&$$

$$e \in Exp ::= 1 \mid \texttt{true} \mid e_1 \oplus e_2$$

Stochastic Grammars

$$\boxed{\oplus \in \mathit{Binop} ::= + \mid \&\&}$$

$$e \in \mathit{Exp} ::= 1 \mid \texttt{true} \mid e_1 \oplus e_2$$

Stochastic Grammars

$$\oplus \in \mathit{Binop} ::= + \mid \&\&$$

$$e \in \mathit{Exp} ::= 1 \mid \texttt{true} \mid e_1 \oplus e_2$$

Stochastic Grammars

$$\oplus \in \mathit{Binop} ::= +^{0.3} \mid \&\&^{0.7}$$

$$e \in \mathit{Exp} ::= 1^{0.4} \mid \mathtt{true}^{0.4} \mid e_1 \oplus e_2{}^{0.2}$$

Stochastic Grammars

$$\oplus \in \mathit{Binop} ::= +^{0.3} \mid \&\&^{0.7}$$

$$e \in \mathit{Exp} ::= \boxed{1^{0.4}} \mid \texttt{true}^{0.4} \mid e_1 \oplus e_2^{\;0.2}$$

$\texttt{e} = \boxed{\texttt{1}}$

Stochastic Grammars

$$\oplus \in \mathit{Binop} ::= +^{0.3} \mid \&\&^{0.7}$$

$$e \in \mathit{Exp} ::= 1^{0.4} \mid \texttt{true}^{0.4} \mid \boxed{e_1 \oplus e_2^{\;0.2}}$$

$$e = \boxed{\_\ \ \_\ \ \_}$$

Stochastic Grammars

$$\oplus \in \mathit{Binop} ::= +^{0.3} \mid \&\&^{0.7}$$

$$e \in \mathit{Exp} ::= 1^{0.4} \mid \boxed{\texttt{true}^{0.4}} \mid e_1 \oplus e_2^{0.2}$$

$e = \boxed{\texttt{true}}\ \_\ \_$

Stochastic Grammars

$$\oplus \in \mathit{Binop} ::= +^{0.3} \mid \boxed{\&\&^{0.7}}$$

$$e \in \mathit{Exp} ::= 1^{0.4} \mid \texttt{true}^{0.4} \mid e_1 \oplus e_2^{0.2}$$

```
e = true && _
```

Stochastic Grammars

$$\oplus \in Binop ::= +^{0.3} \mid \&\&^{0.7}$$

$$e \in Exp ::= 1^{0.4} \mid \boxed{\texttt{true}^{0.4}} \mid e_1 \oplus e_2^{0.2}$$

$\texttt{e = true \&\& }\boxed{\texttt{true}}$

Stochastic Grammars

$$\oplus \in Binop ::= +^{0.3} \mid \&\&^{0.7}$$

$$e \in Exp ::= 1^{0.4} \mid \texttt{true}^{0.4} \mid \boxed{e_1 \oplus e_2 \quad ^{0.2}}$$

$$e = \boxed{\_\ \_\ \_}$$

Stochastic Grammars

$$\oplus \in \mathit{Binop} ::= +^{0.3} \mid \&\&^{0.7}$$

$$e \in \mathit{Exp} ::= \boxed{1^{0.4}} \mid \texttt{true}^{0.4} \mid e_1 \oplus e_2^{0.2}$$

$e = \boxed{1}\ \_\ \_$

Stochastic Grammars

$$\oplus \in Binop ::= \boxed{\overset{0.3}{+}} \mid \overset{0.7}{\&\&}$$

$$e \in Exp ::= \overset{0.4}{1} \mid \overset{0.4}{\texttt{true}} \mid \overset{0.2}{e_1 \oplus e_2}$$

$$\texttt{e} = 1\ \boxed{+}\ \_$$

Stochastic Grammars

$$\oplus \in \mathit{Binop} ::= +^{0.3} \mid \&\&^{0.7}$$

$$e \in \mathit{Exp} ::= 1^{0.4} \mid \boxed{\texttt{true}^{0.4}} \mid e_1 \oplus e_2^{0.2}$$

$$\texttt{e = 1 + }\boxed{\texttt{true}}$$

Type Errors

```
e = 1 + true
```

- Dynamic setting - still a valid test!
- Static setting - tests if typechecker correctly rejects things. Except...
  - No ground truth
  - Most type errors are trivial
  - Most randomly generated programs contain lots of type errors, which can mask each other

Existing Solutions

- All existing solutions that address these concerns suffer from at least one of the following problems:
    - Some generated programs are “accidentally” ill-typed
    - Not all well-typed programs can be generated
    - Fundamentally cannot handle the entire type system
    - Highly specific to language being tested

Existing Solutions

- In all cases, the typechecker is an adversary to be overcome in order to test downstream components
- All implicitly assume the typechecker is correct

Our Approach

Our Approach

- We focus our testing efforts on finding three specific kinds of typechecker bugs:
  - Failure to accept a well-typed program
  - Failure to reject an ill-typed program
  - Inconsistent behavior on *type equivalent* programs
- We have devised general techniques for finding these three kinds of bugs

Viewpoint from Program Analysis (1)

- Failure to accept a well-typed program is a *precision* bug
- While annoying to the programmer, guarantees provided by the type system are preserved

Viewpoint from Program Analysis (2)

- Failure to reject an ill-typed program is a *soundness* bug
- Silent loss of guarantees provided by the type system
- Potentially devastating

Outline

- Background and motivation
- Finding *precision* bugs
- Finding *soundness* bugs
- Finding *consistency* bugs
- Application to Rust
- Results

Finding Precision Bugs

- Intuition: generate guaranteed well-typed programs
- Any rejected programs indicate bugs

Generating Well-Typed Programs

- We use *constraint logic programming* (CLP) for this purpose
- Typing rules can be specified in CLP, and CLP engines can execute them "backwards" to generate programs which are well-typed

Well-Typed Generation Example: System F

System F Highlights

- This is the simply-typed lambda calculus...
    - Higher-order functions
- ...with parametric polymorphism
    - Type variables
- Serves as a simple example
- Despite simplicity, both higher-order functions and type variables fundamentally cannot be handled by prior work

Grammar and Types

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

Grammar and Types

$$
\begin{aligned}
\tau \in \mathit{Type} &::= \boxed{\alpha} \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha . \tau \\
e \in \mathit{Exp} &::= x \mid \lambda x : \tau \, . \, e \mid e_1 \; e_2 \mid \Lambda \alpha \, . \, e \mid e \; \tau
\end{aligned}
$$

Grammar and Types

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

Grammar and Types

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

Grammar and Types

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

Grammar and Types

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

Grammar and Types

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

Grammar and Types

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

Grammar and Types

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

Typing Rules in CLP

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

Typing Rule

CLP Code

```
typing(Gamma, var(X), T) :-
  lookup(Gamma, X, T).
```

Typing Rules in CLP

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

Typing Rule

CLP Code

```
typing(Gamma, var(X), T) :-
  lookup(Gamma, X, T).
```

Typing Rules in CLP

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

Typing Rule

CLP Code

```
typing(Gamma, var(X), T) :-
  lookup(Gamma, X, T).
```

Typing Rules in CLP

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

Typing Rule

CLP Code

```
typing(Gamma, var(X), T) :-
  lookup(Gamma, X, T).
```

Typing Rules in CLP

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

Typing Rule

CLP Code

```
typing(Gamma, var(X), T) :-
   lookup(Gamma, X, T).
```

Typing Rules in CLP

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \boldsymbol{e_1}\ \boldsymbol{e_2} : \tau_2}$$

Typing Rule

CLP Code

```
typing(Gamma, app(E1, E2), T2) :-
  typing(Gamma, E1, arrow(T1, T2)),
  typing(Gamma, E2, T1).
```

Typing Rules in CLP

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

Typing Rule

CLP Code

```
typing(Gamma, app(E1, E2), T2) :-
   typing(Gamma, E1, arrow(T1, T2)),
   typing(Gamma, E2, T1).
```

Typing Rules in CLP

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\boxed{\Gamma} \vdash \boldsymbol{e_1\ e_2} : \tau_2}$$

Typing Rule

CLP Code

```
typing(Gamma, app(E1, E2), T2) :-
  typing(Gamma, E1, arrow(T1, T2)),
  typing(Gamma, E2, T1).
```

Typing Rules in CLP

$$\frac{\boxed{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2} \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \boldsymbol{e_1}\ \boldsymbol{e_2} : \tau_2}$$

Typing Rule

CLP Code

```
typing(Gamma, app(E1, E2), T2) :-
    typing(Gamma, E1, arrow(T1, T2)),
    typing(Gamma, E2, T1).
```

Typing Rules in CLP

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

Typing Rule

CLP Code

```
typing(Gamma, app(E1, E2), T2) :-
  typing(Gamma, E1, arrow(T1, T2)),
  typing(Gamma, E2, T1).
```

Typing Rules in CLP

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

Typing Rule

CLP Code

```
typing(Gamma, app(E1, E2), T2) :-
  typing(Gamma, E1, arrow(T1, T2)),
  typing(Gamma, E2, T1).
```

From Typing Rules to a Generator

- We have implemented a typechecker here
- This can be trivially turned into a generator of well-typed terms, like so (where `?-` indicates what to execute):

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

From Typing Rules to a Generator

- We have implemented a typechecker here
- This can be trivially turned into a generator of well-typed terms, like so (where `?-` indicates what to execute):

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

From Typing Rules to a Generator

- We have implemented a typechecker here
- This can be trivially turned into a generator of well-typed terms, like so (where `?-` indicates what to execute):

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

From Typing Rules to a Generator

- We have implemented a typechecker here
- This can be trivially turned into a generator of well-typed terms, like so (where `?-` indicates what to execute):

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

Take-Home Point

- This generator of well-typed terms can be used to find precision bugs in typecheckers
  - Since everything generated is well-typed, if anything is rejected, it indicates the typechecker under test is buggy under the particular input

Outline

- Background and motivation
- Finding *precision* bugs
- Finding *soundness* bugs
- Finding *consistency* bugs
- Application to Rust
- Results

Finding Soundness Bugs

- Intuition: generate ill-typed programs
- If the typechecker accepts any of them, then we have discovered a bug
- Simple solution: generate syntactically valid programs, and filter out those that happen to be well-typed (which occur rarely)

Finding Soundness Bugs

- A purely syntactic approach results in fairly uninteresting tests
    - They do not exploit information about the underlying type system
    - Tend to be *obviously* ill-typed, so intuitively only the buggiest of typecheckers would let them through

Better Approach

- Generate *almost* well-typed programs, which are ill-typed, but in subtle ways
- Intuitively, one simply negates a single premise of a single typing rule, in a nondeterministic manner
- Based on developer feedback

Almost Well-Typed Generation Example: System F

Typing Rules in CLP

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash \boldsymbol{e_1\ e_2} : \tau_2}$$

Typing Rule

CLP Code

```
typing(Gamma, app(E1, E2), T2) :-
  typing(Gamma, E1, arrow(T1, T2)),
  typing(Gamma, E2, T1).
```

Typing Rules in CLP

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

Typing Rule

CLP Code

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

Outline

- Background and motivation
- Finding *precision* bugs
- Finding *soundness* bugs
- Finding *consistency* bugs
- Application to Rust
- Results

Why Another Type of Bug?

- Theoretically, soundness and precision covers the entire state space
- Finding all possible precision and soundness bugs requires a full-blown typechecker implemented in CLP
  - Lots of work
  - Depending on the language, ground truth may be unclear

Consistency Bugs

- Advantage: full ground truth is not necessary, only an understanding of what constitutes a *type equivalent* program
- This is generally much simpler
- If the typechecker behaves differently on *type equivalent* programs, it indicates a bug
    - Both should be either well-typed or ill-typed

Implementing Consistency Bug Finders

- Basic idea: write a syntax-based generator, using traditional fuzzing techniques
- Pass the output of this generator through a series of rewrite rules
- Ensure that both the input and the output to the rewrite rules behave the same

Outline

- Background and motivation
- Finding *precision* bugs
- Finding *soundness* bugs
- Finding *consistency* bugs
- Application to Rust
- Results

Why Rust?

- A real language with a rapidly growing user base (over 3,300 packages available)
- A sophisticated type system with important guarantees (e.g., memory safety without GC)
- No formal semantics, or even an informal specification
    - Worked closely with Rust development team

Key Rust Type System Features

- Parametric polymorphism
- Generics
- Typeclasses
- Associated types
- Affine types
- Borrowing (reference types)

Testing Methodology

- Handling all of the language with one fuzzer is extremely difficult
- Simpler approach: develop a series of fuzzers which handle subsets of the language
    - Use different techniques for each fuzzer

Outline

- Background and motivation
- Finding *precision* bugs
- Finding *soundness* bugs
- Finding *consistency* bugs
- Application to Rust
- Results

Results

- 18 bugs found across all categories
- 14 confirmed by developers
- Includes a specification-level bug, where a program was legally considered both ill-typed and well-typed
- This work preceded a massive overhaul of the typechecker and overall type system

Conclusions

- We identify three general kinds of typechecker bugs
- We describe automated techniques for finding each of these three kinds of bugs
- We apply these ideas to the Rust programming language, finding 14 confirmed bugs, all of which either have or are being addressed