# RUSTY VARIATION

(or, Deadlock-free sessions with failure in Rust)

by Wen Kokke

# A Tale of Four Examples

# Exceptional GV

(by Fowler et al.)

Looks like this:

$$
\begin{array}{l}
\textbf{let } s = \textbf{fork}(\lambda(s : !\mathbf{1}.\,\mathrm{End}).\\
\qquad \textbf{let } s = \textbf{send}((), s)\\
\qquad \textbf{close}(s)\\
)\\
\textbf{let } ((), s) = \textbf{recv}(s)\\
\textbf{close}(s)
\end{array}
$$

# Rusty Variation

(by me)

Looks like this:

```
let s = fork!(move |s: Send<(), End>| {
    let s = send((), s)?;
    close(s)
});
let ((), s) = recv(s)?;
close(s)
```

# I know, the fonts are very different

# ROADMAP

- » talk about Exceptional GV
- » talk about Rusty Variation
- » what are the differences?
- » what are the similarities?

# EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$\bullet \left( \begin{array}{l} \textbf{let } s = \textbf{fork}(\lambda(s : !1.\mathrm{End}). \\ \qquad \textbf{let } s = \textbf{send}((), s) \\ \qquad \textbf{close}(s) \\ ) \\ \textbf{let } ((), s) = \textbf{recv}(s) \\ \textbf{close}(s) \end{array} \right)$$

We mark the main thread with a $\bullet$

Next we evaluate the fork instruction

# EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{l} \bullet \begin{pmatrix} \textbf{let } s = a \\ \textbf{let } ((), s) = \textbf{recv}(s) \\ \textbf{close}(s) \end{pmatrix} \quad \parallel \\ \circ \begin{pmatrix} \textbf{let } s = \textbf{send}((), b) \\ \textbf{close}(s) \end{pmatrix} \quad \parallel \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) \end{array}\right)$$

This forks off the process and allocates a buffer
Next we evaluate the let binding

# EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{l} \bullet \begin{pmatrix} \mathbf{let}\ ((), s) = \mathbf{recv}(a) \\ \mathbf{close}(s) \end{pmatrix} \quad \parallel \\ \circ \begin{pmatrix} \mathbf{let}\ s = \mathbf{send}((), b) \\ \mathbf{close}(s) \end{pmatrix} \quad \parallel \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) \end{array}\right)$$

The receive instruction blocks on the empty buffer
Next we evaluate the send instruction

# EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$
(\nu a)(\nu b)\left(\begin{array}{l}
\bullet \begin{pmatrix} \textbf{let } ((), s) = \textbf{recv}(a) \\ \textbf{close}(s) \end{pmatrix} \parallel \\
\circ \begin{pmatrix} \textbf{let } s = b \\ \textbf{close}(s) \end{pmatrix} \parallel \\
a((), \epsilon) \leftrightsquigarrow b(\epsilon)
\end{array}\right)
$$

This moves the value to the buffer

Next we evaluate the let binding

# Exceptional GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{l} \bullet \begin{pmatrix}\textbf{let } ((), s) = \textbf{recv}(a) \\ \textbf{close}(s)\end{pmatrix} \quad \parallel \\ \circ\,(\,\textbf{close}(b)\,) \quad \parallel \\ a((), \epsilon) \leftrightsquigarrow b(\epsilon) \end{array}\right)$$

The close instruction blocks (it is synchronous)
Next we evaluate the receive instruction

# Exceptional GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{l} \bullet \begin{pmatrix}\textbf{let}\ ((), s) = ((), a) \\ \textbf{close}(s)\end{pmatrix} \parallel \\ \circ \left(\textbf{close}(b)\right) \parallel \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) \end{array}\right)$$

This moves the value to the main thread
Next we evaluate the let binding

# EXCEPTIONAL GV

Let's see how our example EGV program executes!

$$(\nu a)(\nu b)\left(\begin{array}{l} \bullet\,(\,\mathbf{close}(a)\,) \quad \parallel \\ \circ\,(\,\mathbf{close}(b)\,) \quad \parallel \\ a(\epsilon)\leftrightsquigarrow b(\epsilon) \end{array}\right)$$

The close instructions are no longer blocked

(The buffer is empty and there is a close instruction waiting on either side)

Next we evaluate the close instructions

# Exceptional GV

Let's see how our example EGV program executes!

$$\bullet()$$

Fin

# RUSTY VARIATION

What about our Rust program?

```
let s = fork!(move |s: Send<(), End>| {
    let s = send((), s)?;
    close(s)
});
let ((), s) = recv(s)?;
close(s)
```

# Rusty Variation

```
let s = fork!(move |s: Send<(), End>| {
    let s = send((), s)?;
    close(s)
});
let ((), s) = recv(s)?;
close(s)
```

# Rusty Variation

```
let (s, here) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let s = send((), s)?;
        close(s)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let s = here
let ((), s) = recv(s)?;
close(s)
```

# Rusty Variation

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

# Rusty Variation

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

# RUSTY VARIATION

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

# RUSTY VARIATION

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

# RUSTY VARIATION

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

# RUSTY VARIATION

```
let (b, a) = <Send<(), End> as Session>::new();
std::thread::spawn(move || {
    let r = (move || -> Result<_, Box<Error>> {
        let b = send((), b)?;
        close(b)
    })();
    match r {
        Ok(_) => (),
        Err(e) => panic!("{}", e.description()),
    }
});
let ((), a) = recv(a)?;
close(a)
```

# Sounds familiar?

# LET'S TALK ABOUT ERRORS

# Exceptional GV

(by Fowler et al.)

Looks like this:

$$
\begin{aligned}
&\textbf{let } s = \textbf{fork}(\lambda(s : !1.\,\text{End}).\\
&\qquad \textbf{cancel}(s)\\
&)\\
&\textbf{let } ((), s) = \textbf{recv}(s)\\
&\textbf{close}(s)
\end{aligned}
$$

# Rusty Variation

(by me)

Looks like this:

```
let s = fork!(move |s: Send<(), End>| {
    cancel(s)
});
let ((), s) = recv(s)?;
close(s)
```

# I know, the fonts are very different

# Exceptional GV

Let's see how EGV handles errors!

$$\bullet \left( \begin{array}{l} \textbf{let } s = \textbf{fork}(\lambda(s : !1.\mathrm{End}). \\ \qquad \textbf{cancel}(s) \\ ) \\ \textbf{let } ((), s) = \textbf{recv}(s) \\ \textbf{close}(s) \end{array} \right)$$

We mark the main thread with a $\bullet$

Next we evaluate the fork instruction

# Exceptional GV

Let's see how EGV handles errors!

$$(\nu a)(\nu b)\left(\begin{array}{l} \bullet\left(\begin{array}{l}\mathbf{let}\ s = a\\ \mathbf{let}\ ((), s) = \mathbf{recv}(s)\\ \mathbf{close}(s)\end{array}\right) \quad \parallel \\ \circ\,(\mathbf{cancel}(b)) \quad \parallel \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) \end{array}\right)$$

This forks off the process and allocates a buffer

Next we evaluate the let binding

# EXCEPTIONAL GV

Let's see how EGV handles errors!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet\begin{pmatrix}\textbf{let}\ ((), s) = \textbf{recv}(a) \\ \textbf{close}(s)\end{pmatrix} & \parallel \\ \circ\,(\textbf{cancel}(b)) & \parallel \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) & \end{array}\right)$$

The receive instruction blocks on the empty buffer

Next we evaluate the cancel instruction

# EXCEPTIONAL GV

Let's see how EGV handles errors!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet\begin{pmatrix}\textbf{let}\ ((), s) = \textbf{recv}(a) \\ \textbf{close}(s)\end{pmatrix} & \parallel \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) & \parallel \\ \lightning a & \end{array}\right)$$

This cancels the session and creates a zapper thread
Next we evaluate the receive instruction

# Exceptional GV

Let's see how EGV handles errors!

$$(\nu a)(\nu b)\left(\begin{array}{l} \bullet\begin{pmatrix}\textbf{let}\ ((), s) = \textbf{raise} \\ \textbf{close}(s)\end{pmatrix} \quad \parallel \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) \quad \parallel \\ \lightning b \quad \parallel \\ \lightning a \end{array}\right)$$

Receiving on a channel raises an exception if the other endpoint is cancelled

# Exceptional GV

Let's see how EGV handles errors!

$$(\nu a)(\nu b)\left(\begin{array}{ll} \bullet\, \textbf{halt} & \parallel \\ a(\epsilon) \leftrightsquigarrow b(\epsilon) & \parallel \\ \lightning a & \parallel \\ \lightning b & \end{array}\right)$$

An uncaught exception turns into halt
Next we garbage collect the buffer

# Exceptional GV

Let's see how EGV handles errors!

- **halt**

Fin

# Rusty Variation

What about the Rust library?

```
let s = fork!(move |s: Send<(), End>| {
    cancel(s)
});
let ((), s) = recv(s)?;
close(s)
```

# RUSTY VARIATION

For that, let's look at how cancel is implemented:

```
fn cancel<T>(x: T) -> Result<(), Box<Error>> {
    Ok(())
}
```

Wait, what happened to x?

It went out of scope!

# Rusty Variation

What happens when a channel x leaves scope unused?

- destructor is called
- values in buffer are deallocated
- destructors for values in buffer are called
- buffer is marked as DISCONNECTED
- calling recv on DISCONNECTED buffer returns Err

# Sounds familiar?

# WHAT ARE THE DIFFERENCES?

» try/catch vs. error monad

(using the "$\textbf{try}\ L\ \textbf{as}\ x\ \textbf{in}\ N\ \textbf{otherwise}\ M$" instruction)

» explicit close vs. implicit close

```
fn close(s: End) -> Result<(), Box<Error>> {
    Ok(()) // `End` doesn't have a buffer
}
```

» explicit cancellation vs. implicit cancellation

(what happens if we forget to complete a session?)

# WHAT ARE THE DIFFERENCES?

- simply-typed linear lambda calculus vs. Rust

  this means we have:

  - no recursion vs. general recursion
  - lock freedom vs. deadlock freedom
  - etc.

# How can we get deadlocks in Rusty Variation?

- by using mem::forget

```
let s = fork!(move |s: Send<(), End>| {
    mem::forget(s);
    Ok(())
});
let ((), s) = recv(s)?;
close(s)
```

- by storing channels in manually managed memory and not cleaning up

# What are the similarities?

» in theory, everything else?

» can we prove it?

“doesn't Rust have formal semantics?
I heard so much about RustBelt!

no.

RustBelt formalises elaborated Rust and
doesn't support many features we depend on.

# What are the similarities?

- in theory, everything else?
- can we prove it? no.
- can we test it?

```rust
#[test]
fn ping_works() {
    assert!(|| -> Result<(), Box<Error>> {

        // ...insert example here...

    }().is_ok()); // it actually is!
}
```

# What are the similarities?

» in theory, everything else?

» can we prove it? no.

» can we test it? yes.

» can we <u>properly</u> test it?

# Testing Rusty Variation

```
(x) generate random EGV term
( ) compile EGV term to Rust
( ) run Rust term and log trace
( ) check trace with EGV term
```

# The following is an ad for FEAT/NEAT

by Claessen, Duregård, & Pałka

# I wanna make stuff like this...

$$
\begin{aligned}
&\mathbf{let}\ s = \mathbf{fork}(\lambda(s : !\mathbf{1}.\,\mathrm{End}).\\
&\qquad \mathbf{let}\ s = \mathbf{send}((), s)\\
&\qquad \mathbf{close}(s)\\
&)\\
&\mathbf{let}\ ((), s) = \mathbf{recv}(s)\\
&\mathbf{close}(s)
\end{aligned}
$$

# Let's not get ahead of ourselves, though...

$$\lambda(f : \mathbf{0} \multimap \mathbf{0}).\, \lambda(x : \mathbf{0}).\, f\ x$$

$$\lambda(f : \mathbf{0} \multimap \mathbf{0}).\, \lambda(g : \mathbf{0} \multimap \mathbf{0}).\, \lambda(x : \mathbf{0}).\, f\ (g\ x)$$

$$(\lambda(f : \mathbf{0} \multimap \mathbf{0}).\, f)\ (\lambda(x : \mathbf{0}).\, x)$$

$$(\lambda(g : (\mathbf{0} \multimap \mathbf{0}) \multimap (\mathbf{0} \multimap \mathbf{0})).\, g)\ (\lambda(f : \mathbf{0} \multimap \mathbf{0}).\, f)$$

# Let's try...

# QuickCheck: Automatic testing of Haskell programs

[ bsd3, library, testing ] [ Propose Tags ]

QuickCheck is a library for random testing of program properties. The programmer provides a specification of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in Haskell, using combinators provided by QuickCheck. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators.

# QuickCheck

Mo[illegible]eck's functionality i[illegible]xported by the [illegible]ain Test.Quick[illegible]du[illegible] The main exception is the monad[illegible] p[illegible]rty test[illegible]ibrary in Test.Quic[illegible]eck.Monadic.

I[illegible] y[illegible]u are new [illegible] Qui[illegible]heck [illegible]ou c[illegible] try l[illegible] at [illegible] f[illegible]wing [illegible]urces:

- The [illegible]ffici[illegible] uic[illegible]eck [illegible]nua[illegible] t's a[illegible] out-of-[illegible]e [illegible]ome [illegible]ils and doe[illegible] t cov[illegible] nev[illegible] QuickCh[illegible] feature[illegible] u[illegible] [illegible]ood [illegible]ice.
- https[illegible]egriffs.com/posts/2017-01-14-design-use-quickcheck.html, a detailed tutorial written by a user of QuickCheck.

The quickcheck-instances companion package provides instances for types in Haskell Platform packages at the cost of additional dependencies.

---

[Skip to Readme]

## Modules

[Index] [Quick Jump]

*Test*

## Versions

1.0, 1.1.0.0, 1.2.0.0, 1.2.0.1, 2.1, 2.1.0.1, 2.1.0.2, 2.1.0.3, 2.1.1, 2.1.1.1, 2.1.2, 2.2, 2.3, 2.3.0.1, 2.3.0.2, 2.4, 2.4.0.1, 2.4.1, 2.4.1.1, 2.4.2, 2.5, 2.5.1, 2.5.1.1, 2.6, 2.7, 2.7.1, 2.7.2, 2.7.3, 2.7.4, 2.7.5, 2.7.6, 2.8, 2.8.1,

# Intermission

"What is this QuickCheck you speak of?"

# QuickCheck 101

## You write...

```
import Test.QuickCheck

prop_revapp :: [Int] -> [Int] -> Bool
prop_revapp xs ys = reverse (xs ++ ys) == reverse ys ++ reverse xs
```

## You test...

```
>>> quickCheck prop_revapp
+++ OK, passed 100 tests.
```

# QuickCheck 101

```haskell
instance Arbitrary Int where
    arbitrary = choose (minBound, maxBound)
              -- ^ pick number between -2^29 and 2^29-1

instance Arbitrary a => Arbitrary [a] where
    arbitrary = do n <- arbitrary
                   replicateM n arbitrary
              -- ^ pick arbitrary length n
              --   pick n things of type a
```

So now we all know exactly how QuickCheck works...

# My good plan

1. make some programs
2. run them programs
3. compile them to Rust
4. run them in Rust
5. see if they same

# QuickCheck, some programs please?

"There is no generic arbitrary implementation included because we don't know how to make a high-quality one. If you want one, consider using the `testing-feat` or `generic-random` packages."

— xoxo QuickCheck

# Fine! I'll write one myself! 😤

```haskell
type Name = String

data Term = Var Name
          | Lam Name Term
          | App Term Term

instance Arbitrary Term where
    arbitrary = oneof
        [ Var <$> arbitrary
        , Lam <$> arbitrary <*> arbitrary
        , App <$> aribtrary <*> arbitrary ]
```

# Fine! I'll write one myself!

# Lam ">h" (Var "\EOT\NAKW")

```haskell
type Name = String

data Term = Var Name
          | Lam Name Term
          | App Term Term

instance Arbitrary Term where
    arbitrary = oneof
        [ Var <$> arbitrary
        , Lam <$> arbitrary <*> arbitrary
        , App <$> aribtrary <*> arbitrary ]
```

# ... Oh, right

# Uhh, I guess I'll do some thinking... 😫

```
data Z            -- Z has no elements

data S n          -- S n has |n| + 1 elements
    = FZ          -- e.g. TwoOfFour :: S (S (S (S Z)))
    | FS n        --      TwoOfFour = FS (FS FZ)

data Term n                          -- "every term is
    = Var n                          -- well-scoped,
    | Lam (Term (S n))               -- so no more
    | App (Term n) (Term n)          -- nonsense."
```

# How do I random these? 🤔

```
instance Arbitrary Z where
    arbitrary = oneof [] -- a blatant lie

instance Arbitrary n => Arbitrary (S n) where
    arbitrary = oneof [ pure FZ , FS <$> arbitrary ]

instance Arbitrary n => Arbitrary (Term n) where
    arbitrary = oneof
        [ Var <$> arbitrary
        , Lam <$> arbitrary
        , App <$> aribtrary <*> arbitrary ]
```

# How do I random these? 🤔

```haskell
instance Arbitrary Z where
    arbitrary = oneof [] -- a blatant lie

instance Arbitrary n => Arbitrary (S n) where
    arbitrary = oneof [ pure FZ , FS <$> arbitrary ]

instance Arbitrary n => Arbitrary (Term n) where
    arbitrary = oneof
        [ Var <$> arbitrary
        , Lam <$> arbitrary
        , App <$> aribtrary <*> arbitrary ]
```

**Lam (Lam (Var (FS FZ)))**

...

**Woohoo~!** ✨

# But types?

# Let's add some types...

```
data Type
     = Void
     | Type :-> Type

data Term n
     = Var n
     | Lam (Term (S n))
     | App (Term n) (Term n) Type -- this is new!

check :: [Type] -> Type -> Term n -> Bool
check env a         (Var n)     = lookup env n == a
check env (a :-> b) (Lam t)     = check (a : env) b t
check env b         (App f s a) = check env (a :-> b) f && check env a s
check _   _         _           = False
```

# Only the well-typed ones plz?

```haskell
instance Arbitrary Type where
    arbitrary = oneof
        [ pure Void
        , (:->) <$> arbitrary <*> arbitrary ]

newtype WellTyped n = WellTyped (Term n)

instance Arbitrary WellTyped Z where
    arbitrary = do
        a <- arbitrary -- an arbitrary type
        t <- arbitrary -- an arbitrary *closed* term
        if check [] t a then WellTyped t else arbitrary
```

# Only the well-typed ones plz?

```
instance Arbitrary Type where
    arbitrary = oneof
        [ pure Void
        , (:->) <$> arbitrary <*> arbitrary ]

newtype WellTyped n = WellTyped (Term n)

instance Arbitrary WellTyped Z where
    arbitrary = do
        a <- arbitrary -- an arbitrary type
        t <- arbitrary -- an arbitrary *closed* term
        if check [] t a then WellTyped t else arbitrary
```

...

Uh? 😯

# Only the well-typed ones plz?

```haskell
instance Arbitrary Type where
    arbitrary = oneof
        [ pure Void
        , (:->) <$> arbitrary <*> arbitrary ]

newtype WellTyped n = WellTyped (Term n)

instance Arbitrary WellTyped Z where
    arbitrary = do
        a <- arbitrary -- an arbitrary type
        t <- arbitrary -- an arbitrary *closed* term
        if check [] t a then WellTyped t else arbitrary
```

...

What's going on? 😮

# Only the well-typed ones plz?

```haskell
instance Arbitrary Type where
    arbitrary = oneof
        [ pure Void
        , (:->) <$> arbitrary <*> arbitrary ]

newtype WellTyped n = WellTyped (Term n)

instance Arbitrary WellTyped Z where
    arbitrary = do
        a <- arbitrary -- an arbitrary type
        t <- arbitrary -- an arbitrary *closed* term
        if check [] t a then WellTyped t else arbitrary
```

Why is nothing happening? 😦

# Only the well-typed ones plz?

```
instance Arbitrary Type where
    arbitrary = oneof
        [ pure Void
        , (:->) <$> arbitrary <*> arbitrary ]

newtype WellTyped n = WellTyped (Term n)

instance Arbitrary WellTyped Z where
    arbitrary = do
        a <- arbitrary -- an arbitrary type
        t <- arbitrary -- an arbitrary *closed* term
        if check [] t a then WellTyped t else arbitrary
```

...

Halp?! 😱

# Blech!

I guess I'll do some research...

# What proportion of lambda terms is typeable?

## A Natural Counting of Lambda Terms

Maciej Bendkowski[1] (✉), Katarzyna Grygiel[1],
Pierre Lescanne[2], and Marek Zaionc[1]

[1] Faculty of Mathematics and Computer Science,
Theoretical Computer Science Department, Jagiellonian University,
ul. Prof. Łojasiewicza 6, 30–348 Kraków, Poland
{bendkowski,grygiel,zaionc}@tcs.uj.edu.pl
[2] École Normale Supérieure de Lyon,
LIP (UMR 5668 CNRS ENS Lyon UCBL INRIA),
University of Lyon, 46 Allée d'Italie, 69364 Lyon, France
pierre.lescanne@ens-lyon.fr

**Abstract.** We study the sequence of numbers corresponding to $\lambda$-terms of given size in the model based on de Bruijn indices. It turns out that the sequence enumerates also two families of binary trees, i.e. black-white and zigzag-free ones. We provide a constructive proof of this fact by exhibiting

# What proportion of lambda terms is **strongly normalising?**

"Corollary 1.
Asymptotically almost no λ-term is strongly normalizing."

-- xoxo Bendkowski, Grygiel, Lescanne, and Zaionc

# Can we solve this through engineering?

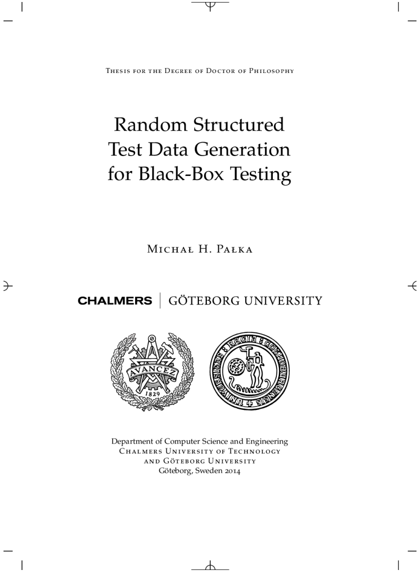
## Generating Constrained Random Data with Uniform Distribution

Koen Claessen, Jonas Duregård, and Michał H. Pałka

Chalmers University of Technology
{koen,jonas.duregard,michal.palka}@chalmers.se

**Abstract.** We present a technique for automatically deriving test data generators from a predicate expressed as a Boolean function. The distribution of these generators is uniform over values of a given size. To make the generation efficient we rely on laziness of the predicate, allowing us to prune the space of values quickly. In contrast, implementing test data generators by hand is labour intensive and error prone. Moreover, handwritten generators often have an unpredictable distribution of values, risking that some values are arbitrarily underrepresented. We also present a variation of the technique where the distribution is skewed in a limited and predictable way, potentially increasing the performance. Experimental evaluation of the techniques shows that the uniform derived generators are much easier to define than hand-written ones, and their performance, while lower, is adequate for some realistic applications.

# Can we solve this through engineering?



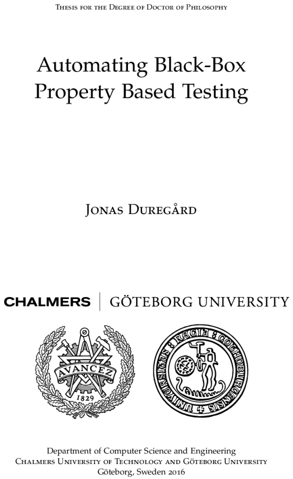
Thesis for the Degree of Doctor of Philosophy

Random Structured
Test Data Generation
for Black-Box Testing

Michał H. Pałka

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology
and Göteborg University
Göteborg, Sweden 2014



Thesis for the Degree of Doctor of Philosophy

Automating Black-Box
Property Based Testing

Jonas Duregård

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2016

# So what is their trick?

# Bag of tricks

1. some gross stuff to **ensure sharing**
   (implemented in the `size-based` package)
2. some **DSL magic** for building enumerations
   (implemented in the `testing-feat` package)
3. some gross stuff to **filter ill-typed terms eagerly**
   (implemented in the `lazy-search` package)

# QuickCheck, some programs please?

"There is no generic arbitrary implementation included because we don't know how to make a high-quality one. If you want one, consider using the `testing-feat` or `generic-random` packages."

— xoxo QuickCheck

# Gross stuff to ensure sharing

(hint: it's encapsulated global mutable state)

# DSL magic for building enumerations

```
instance Enumerable Z where
    enumerate = datatype [] -- a blatant truth

instance Enumerable n => Enumerable (S n) where
    enumerate = datatype [ v0 FZ , v1 FS ]

instance Enumerable Type where
    enumerate = datatype [ v0 Void , v2 (:->) ]

instance Enumerable n => Enumerable (Term n) where
    enumerate = datatype [ pay (v1 Var) , pay (v1 Lam) , pay (v3 App) ]
```

# Does it work out of the box?

```
-- get me all programs of size <30
$ eleanor --system Untyped --action Print --size 30
[Lam (Var FZ), Lam (Lam (Var FZ)), Lam (Lam (Var (FS (FZ)))), ...]

-- how many programs of size <30?
$ eleanor --system Untyped --action Count --size 30
796494839 1145

-- how many programs of size <100?
$ eleanor --system Untyped --action Count --size 100
45037877201949315009360216882885664284506471988998311319 20
```

# Does it work out of the box?

```
-- how many programs of size <1000?
$ eleanor --system Untyped --size 1000
308979047539797286389554754656050850905240507708427967498701817852887971931069975365901
857378119631500575402859069294978611884417142648912870521418834178736010885629562442174
695729552893817244891920582785029398882622008238200608644806387090253102487903461107900
446985363433164099802667368836306482954336643903824771835185388183129889962918463489147
669085392503510337274432408608493215807279736697555590998870222330656848190305130272295
748823658429313198623977474018608312268715019965824283441864212858719037406270777784320
128035445486523339972120044617149804509803809721945756672127484790222562203093028297330
701810553080361603375463934103265024019533365037819232420615636268119286995638542364078
581194561105664479452966258068391627683565675385447131617537498143916191855677543179164
384243554806966886472148143594689568030174613831597761325860

real	1m 26.740s
user	1m 23.087s
sys	0m  1.216s
```

# Gross stuff to filter ill-typed terms eagerly

```
univ :: (a -> Bool) -> (b -> a) -> Maybe Bool
univ pred val = unsafePerformIO $
    Just (pred (val undefined)) `catch` \err -> Nothing

-- will this program ever be well-typed?
> univ (check [] Void) (\hole -> Lam hole)
Just False -- no

-- will this program ever be well-typed?
> univ (check [] (Void :-> Void)) (\hole -> Lam hole)
Nothing -- dunno?
```

# Gross stuff to filter ill-typed terms eagerly

```
univ :: (a -> Bool) -> (b -> a) -> Maybe Bool
univ pred val = unsafePerformIO $
  Just (pred (val undefined)) `catch` \err -> Nothing

-- will this program ever be well-typed?
> univ (check [] (Void ...)) (\hole -> Lam hole)
Just False -- no

-- will this program ever be well-typed?
> univ (check [] (Void :-> Void)) (\hole -> Lam hole)
Nothing -- dunno?
```

**Only works if your predicate is eager!**

# Does it work out of the box?

```
-- get me the programs of type `Void :-> Void` and size <30!
$ eleanor --system SimplyTyped --action Print --size 30
[Lam (Var FZ), Lam (App (Lam (Var FZ)) (Var FZ) Void), ...]

-- how many programs of type `Void :-> Void` and size <30?
$ eleanor --system SimplyTyped --action Count --size 30
11369362

real   6m 31.701s  -- does not look as good
user   6m 25.991s  -- slower by a magnitude
sys    0m  3.950s  -- but better than anything I've written
```

# Problem solved! 🎉

I made a lie 😭

# This is linear[1]

$$
\begin{aligned}
&\textbf{let } s = \textbf{fork}(\lambda(s : !1.\,\mathrm{End}).\\
&\qquad \textbf{let } s = \textbf{send}((), s)\\
&\qquad \textbf{close}(s)\\
&)\\
&\textbf{let } ((), s) = \textbf{recv}(s)\\
&\textbf{close}(s)
\end{aligned}
$$

[1] (i.e. variables must be used exactly once)

# This is affine[2]

```
let s = fork!(move |s: Send<(), End>| {
    let s = send((), s)?;
    close(s)
});
let ((), s) = recv(s)?;
close(s)
```

[2] (i.e. variables can be used at most once)

# This is neither[3]

```haskell
data Type
    = Void
    | Type :-> Type

data Term n
    = Var n
    | Lam (Term (S n))
    | App (Term n) (Term n) Type -- this isn't new anymore

check :: [Type] -> Type -> Term n -> Bool
check env a          (Var n)     = lookup env n == a
check env (a :-> b) (Lam t)     = check (a : env) b t
check env b          (App f s a) = check env (a :-> b) f && check env a s
check _   _          _           = False
```

[3] (i.e. variables can do *whatever they want!* 😱)

I sorry 😭

# Idea! 💡

Generate programs, then take the linear ones!

# What proportion of all programs is linear?

*Universal Logic Corner*

## How big is BCI fragment of BCK logic

KATARZYNA GRYGIEL, PAWEł M. IDZIAK and MAREK ZAIONC
*Department of Theoretical Computer Science, Faculty of Mathematics and Computer Science, Jagiellonian University, Łojasiewicza 6, 30-348 Kraków, Poland.*
*E-mail: grygiel@tcs.uj.edu.pl; idziak@tcs.uj.edu.pl; zaionc@tcs.uj.edu.pl*

### Abstract

We investigate quantitative properties of BCI and BCK logics. The first part of the article compares the number of formulas provable in BCI versus BCK logics. We consider formulas built on implication and a fixed set of $k$ variables. We investigate the proportion between the number of such formulas of a given length $n$ provable in BCI logic against the number of formulas of length $n$ provable in richer BCK logic. We examine an asymptotic behaviour of this fraction when length $n$ of formulas tends to infinity. This limit gives a probability measure that randomly chosen BCK formula is also provable in BCI . We prove that this probability tends to zero as the number of variables tends to infinity. The second part of the article is devoted to the number of lambda terms representing proofs of BCI and BCK logics. We build a proportion between number of such proofs of the same length $n$ and we investigate asymptotic behaviour of this proportion when length of proofs tends to infinity. We demonstrate that with probability 0 a randomly chosen BCK proof is also a proof of a BCI formula.

*Keywords*: BCK and BCI logics, asymptotic probability in logic, analytic combinatorics.

**What proportion of affine programs is linear?**

"Theorem 42.
The density of BCI terms among BCK terms equals 0."

— xoxo Grygiel, Idiziak, and Zaionc

## Can we solve this through engineering?

"Sometimes you just have to be stupid and try to search an immensely huge search space just 'cuz you can."

— xoxo some A.I. researcher (probably)

# I tried

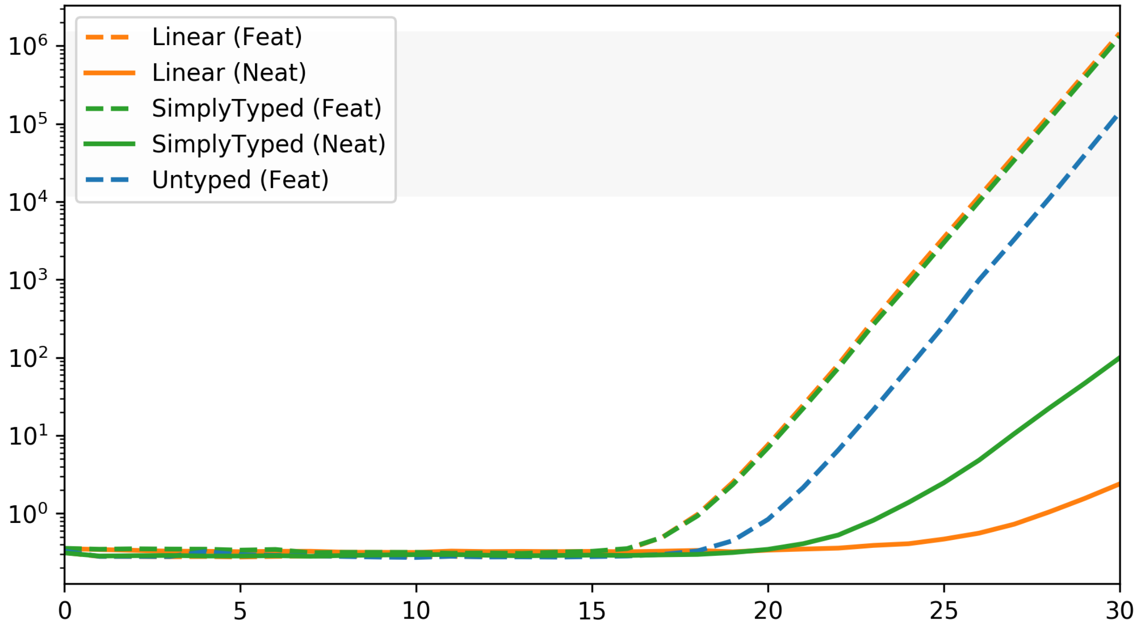
```haskell
check :: Fin => Type -> Term n -> State (Map n Type) Bool
check a (Var x) = do
    env <- get                               -- ...
    modify (delete x)                        -- remove variable
    return $ lookup FZ env == Just a         -- was the type right?
check (a :-> b) (Lam t) = do
    modify (insert FS a . mapKeys FS)        -- insert new variable
    cond1 <- check a t                       -- check body
    env   <- get                             -- ...
    let cond2 = lookup FZ env == Nothing     -- was new variable used?
    modify (mapKeys pred)                    -- restore old variables
    return $ cond1 && cond2
check b (App f s a) = do
    cond1 <- check (a :-> b) f               -- check function
    cond2 <- check a s                       -- check argument
    return $ cond1 && cond2
check _ _ = do return False
```

(That's too much code, Wen! 😨)

# I tried, it's pretty good, actually...

```
-- how many linear programs of type `Void :-> Void` and size <30?
$ eleanor --system Linear --action Count --size 30
9790

real	0m	2.580s
user	0m	2.361s
sys	0m	0.264s
```


Linear (Feat)
Linear (Neat)
SimplyTyped (Feat)
SimplyTyped (Neat)
Untyped (Feat)
$10^6$
$10^5$
$10^4$
$10^3$
$10^2$
$10^1$
$10^0$
0
5
10
15
20
25
30

# Testing Rusty Variation

```
(x) generate random EGV term
( ) compile EGV term to Rust
( ) run Rust term and log trace
( ) check trace with EGV term
```

So, uh...

Where were we?

# How efficient is Rusty Variation?

- buffers are either empty or non-empty
- size of buffers is statically known

  (unless you're sending boxed references)
- each buffer only involves a single allocation
- size of session is statically known

  (but buffers are allocated lazily)
- it's really quite efficient y'all

# RELATED WORK

# session-types

(by Laumann et al.)

- library for session types in Rust
- dibsed the best package name
- embeds LAST[2] in Rust

  (a linear language embedded in an affine one)
- <u>forget to complete a session? segfault!</u>

[2] Linear type theory for asynchronous session types, Gay & Vasconcelos, 2010

# CONCLUSIONS

# RUSTY VARIATION

- » embeds EGV into Rust
- » is unit tested
- » will be QuickChecked
- » is very efficient
- » improves session-types

toot