Where the linear lambdas go

by Wen Kokke


Me, reading "Session Types without Tiers" by Fowler et al.

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


Me, implementing "Session Types without Tiers" in Rust.

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


They look the same.
Do they do the same?

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

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

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


Well that sounds ok.
Maybe we prove?

There's formal semantics for Rust, right?

RustBelt: Securing the Foundations of the Rust Programming Language

RALF JUNG, MPI-SWS, Germany
JACQUES-HENRI JOURDAN, MPI-SWS, Germany
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
DEREK DREYER, MPI-SWS, Germany

66

Rust is a new systems programming language that promises to overcome the seemingly fundamental tradeoff between high-level safety guarantees and low-level control over resource management. Unfortunately, none of Rust's safety claims have been formally proven, and there is good reason to question whether they actually hold. Specifically, Rust employs a strong, ownership-based type system, but then extends the expressive power of this core type system through libraries that internally use unsafe features. In this paper, we give the first formal (and machine-checked) safety proof for a language representing a realistic subset of Rust. Our proof is extensible in the sense that, for each new Rust library that uses unsafe features, we can say what verification condition it must satisfy in order for it to be deemed a safe extension to the language. We have carried out this verification for some of the most important libraries that are used throughout the Rust ecosystem.

CCS Concepts: • **Theory of computation** → Programming logic; Separation logic; Operational semantics;

Additional Key Words and Phrases: Rust, separation logic, type systems, logical relations, concurrency

ACM Reference Format:
Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (January 2018), 34 pages. https://doi.org/10.1145/3158154

1 INTRODUCTION

Systems programming languages like C and C++ give programmers low-level control over resource management at the expense of safety, whereas most other modern languages give programmers safe, high-level abstractions at the expense of control. It has long been a "holy grail" of programming languages research to overcome this seemingly fundamental tradeoff and design a language that offers programmers both high-level safety *and* low-level control.

Rust Distilled: An Expressive Tower of Languages

AARON WEISS, Northeastern University and Inria Paris
DANIEL PATTERSON, Northeastern University
AMAL AHMED, Northeastern University and Inria Paris

Rust represents a major advancement in production programming languages because of its success in bridging the gap between *high-level* application programming and *low-level* systems programming. At the heart of its design lies a novel approach to *ownership* that remains highly programmable. In this talk, we will describe our ongoing work on designing a formal semantics for Rust that captures ownership and borrowing without the details of lifetime analysis. This semantics models a high-level understanding of ownership and as a result is close to source-level Rust (but with full type annotations) which differs from the recent RustBelt effort that essentially models MIR, a CPS-style IR used in the Rust compiler. Further, while RustBelt aims to verify the safety of `unsafe` code in Rust's standard library, we model standard library APIs as primitives, which is sufficient to reason about their behavior. This yields a simpler model of Rust and its type system that we think researchers will find easier to use as a starting point for investigating Rust extensions. Unlike RustBelt, we aim to prove type soundness using *progress and preservation* instead of a Kripke logical relation. Finally, our semantics is a family of languages of increasing *expressive power*, where subsequent levels have features that are impossible to define in previous levels. Following Felleisen, expressive power is defined in terms of *observational equivalence*. Separating the language into different levels of expressive power should provide a framework for future work on Rust verification and compiler optimization.

1 INTRODUCTION

Programming languages have long been divided between "systems" languages, which enable low-level reasoning that has proven critical in writing systems software, and "high-level" languages, which empower programmers with high-level abstractions to write software more quickly and more safely. For many language researchers then, a natural goal has been to try to enable both low-level reasoning and high-level abstractions in one language. To date, the Rust programming language has been the most successful endeavour toward such a goal.

Nevertheless, Rust has also developed something of a reputation for its complexity amongst

No. Not really.

RustBelt: Securing the Foundations of the Rust Programming Language

66

RALF JUNG, MPI-SWS, Germany
JACQUES-HENRI JOURDAN, MPI-SWS, Germany
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
DEREK DREYER, MPI-SWS, Germany

Rust i[illegible]ystems [illegible]mming language that promises to overcome [illegible]ingly fu[illegible]ntal tradeoff betw[illegible]vel saf[illegible]uarantees and low-level control over resource [illegible]ent. U[illegible]ately, none of Ru[illegible]ims [illegible]been fo[illegible]en, and there is good reaso[illegible] they act[illegible] hold. [illegible]ust [illegible]ys a s[illegible]hip-based type system, bu[illegible] the ex[illegible]essive [illegible] of th[illegible] typ[illegible]ough [illegible]ries th[illegible]ternally use unsafe feat[illegible] give [illegible]rst form[illegible] mach[illegible]d) sa[illegible]proof f[illegible] language representing a [illegible]tic sub[illegible] Our [illegible] is exte[illegible]n the se[illegible] for ea[illegible]ew Ru[illegible]rar[illegible] uses unsafe fe[illegible], we can [illegible] veri[illegible]on cond[illegible]t must sa[illegible]rder [illegible]eme[illegible]e extension to [illegible]nguage. [illegible] carr[illegible] this verification for some of the most important libraries that are used throughout the Rust ecosystem.

CCS Concepts: • **Theory of computation** → Programming logic; Separation logic; Operational semantics;

Additional Key Words and Phrases: Rust, separation logic, type systems, logical relations, concurrency

ACM Reference Format:
Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (January 2018), 34 pages. https://doi.org/10.1145/3158154

1 INTRODUCTION

Systems programming languages like C and C++ give programmers low-level control over resource management at the expense of safety, whereas most other modern languages give programmers safe, high-level abstractions at the expense of control. It has long been a "holy grail" of programming languages research to overcome this seemingly fundamental tradeoff and design a language that offers programmers both high-level safety *and* low-level control.

Rust Distilled: An Expressive Tower of Languages

AARON WEISS, Northeastern University and Inria Paris
DANIEL PATTERSON, Northeastern University
AMAL AHMED, Northeastern University and Inria Paris

Rust represents a major advancement in production programming languages because of its success in bridging the gap between *high-level* application programming and *lo*[illegible]*vel* sy[illegible]s programming. At the heart of its design lies a novel approach to *ownership* that remains high[illegible]rogra[illegible]able.

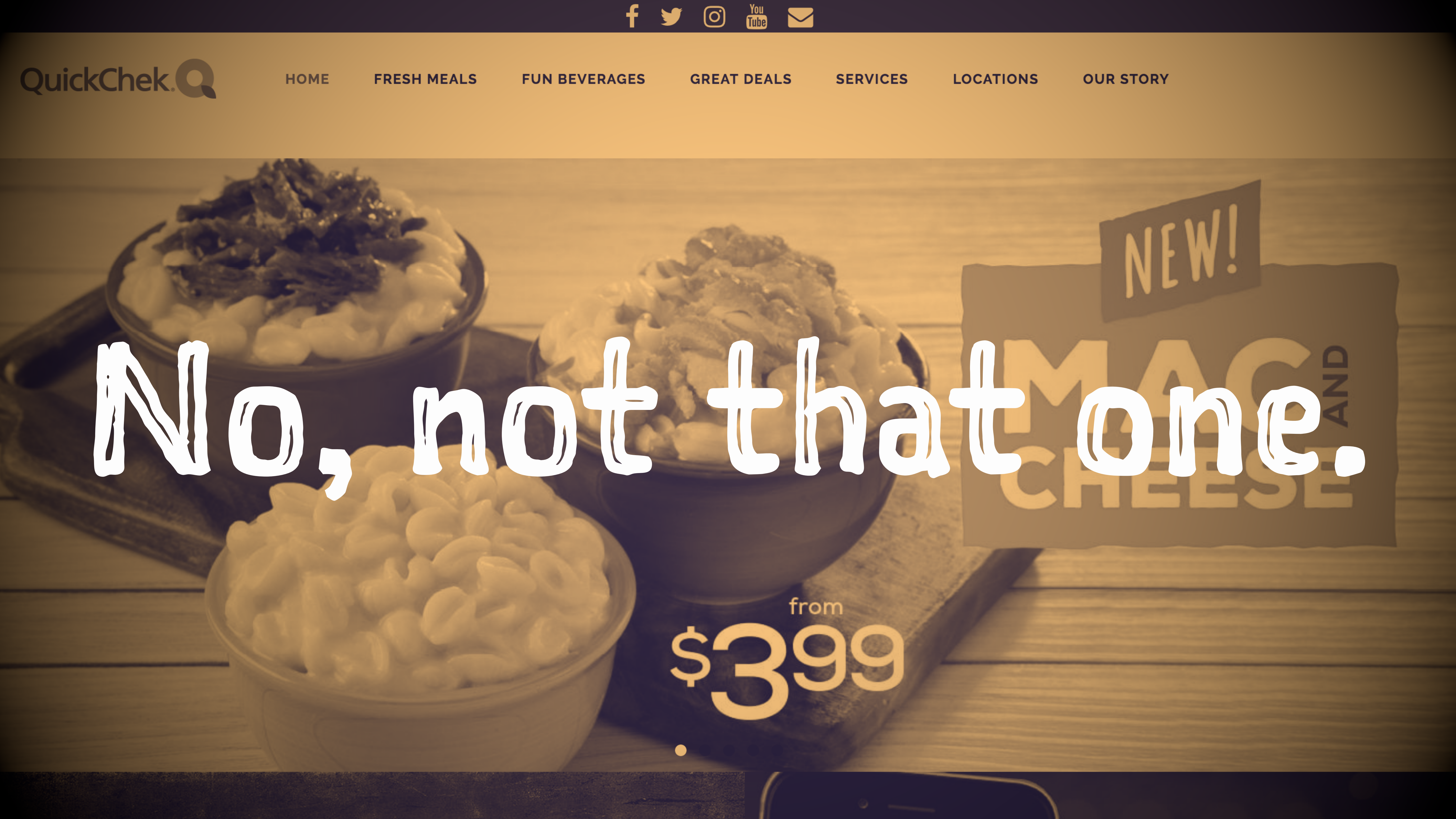
In this talk, we will describe our ongoing work on desi[illegible]g a fo[illegible]l semantics for Rust that captures ownership a[illegible]or[illegible]g w[illegible] de[illegible]etime a[illegible]ysis. T[illegible] sem[illegible]s mod[illegible] high-level understanding [illegible]ip a[illegible] a r[illegible]t is close [illegible]urce-[illegible] Rust [illegible]t wit[illegible] type a[illegible]tations) which differs from [illegible]cent Rus[illegible]hat [illegible] mode[illegible] IR, a [illegible]-sty[illegible] used in [illegible] Rust compiler. Further, whil[illegible]stBelt aim[illegible] verify the s[illegible]y of [illegible]fe c[illegible] in Ru[illegible] stand[illegible]ibrary, [illegible] model standard library APIs [illegible]rimitives, [illegible] is suffici[illegible] to r[illegible] abo[illegible]heir b[illegible]vior. [illegible]ield [illegible]mple[illegible]el of Rust and its [illegible] system tha[illegible]rese[illegible]nd [illegible]er to [illegible]s a st[illegible]r in[illegible]ting Rust extensions. Unlike RustBelt, we aim to prove type soundness using *progress and p*[illegible]*rvation* instead of a Kripke logical relation. Finally, our semantics is a family of languages of increasi[illegible]*xpressive power*, where subsequent levels have features that are impossible to define in previou[illegible]owing Felleisen, expressive power is defined in terms of *observational equivalence*. Separating the language into different levels of expressive power should provide a framework for future work on Rust verification and compiler optimization.

1 INTRODUCTION

Programming languages have long been divided between "systems" languages, which enable low-level reasoning that has proven critical in writing systems software, and "high-level" languages, which empower programmers with high-level abstractions to write software more quickly and more safely. For many language researchers then, a natural goal has been to try to enable both low-level reasoning and high-level abstractions in one language. To date, the Rust programming language has been the most successful endeavour toward such a goal.

Nevertheless, Rust has also developed something of a reputation for its complexity amongst

Let's try QuickCheck?


QuickChek
HOME
FRESH MEALS
FUN BEVERAGES
GREAT DEALS
SERVICES
LOCATIONS
OUR STORY
NEW!
MAC AND CHEESE
from
$399
No, not that one.

Yes, thank you.

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.

Most of QuickCheck's functionality is exported by the main Test.QuickCheck module. The main exception is the monadic property testing library in Test.QuickCheck.Monadic.

If you need to learn QuickCheck, you can try the following resources:

- The official QuickCheck manual. It's a bit out of date in some details and doesn't cover newer QuickCheck features, but is still full of good advice.
- https://begriffs.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,

What is this "QuickCheck" you speak of?

QuickCheck 101

You write...

```haskell
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 102

QuickCheck knows how to make random numbers...

```
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. translate them to Rust
4. run them in Rust
5. see if they same

QuickCheck, please make some programs?

"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!

```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!

```
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
    ]
```

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

Oh no! 😞

Eugh, 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 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?

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

Yay!

But types?

Cool, 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?

```
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?

Halp?!

```
ance A     rary Type where
arbitra     = oneof
    [ p       Void
            <$          ary     arb
    ]

n w ype We  T  ped    = We   ype   Term

ance A     rary          Z     re
arbitrary = do
    a <- arbitrary -- an arbitrary  pe
    t <- arbitrary -- an arbitrar     losed~ term
    if check [] t a then WellTyped t else arbitrary
```

Eugh, I guess I'll do some research

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

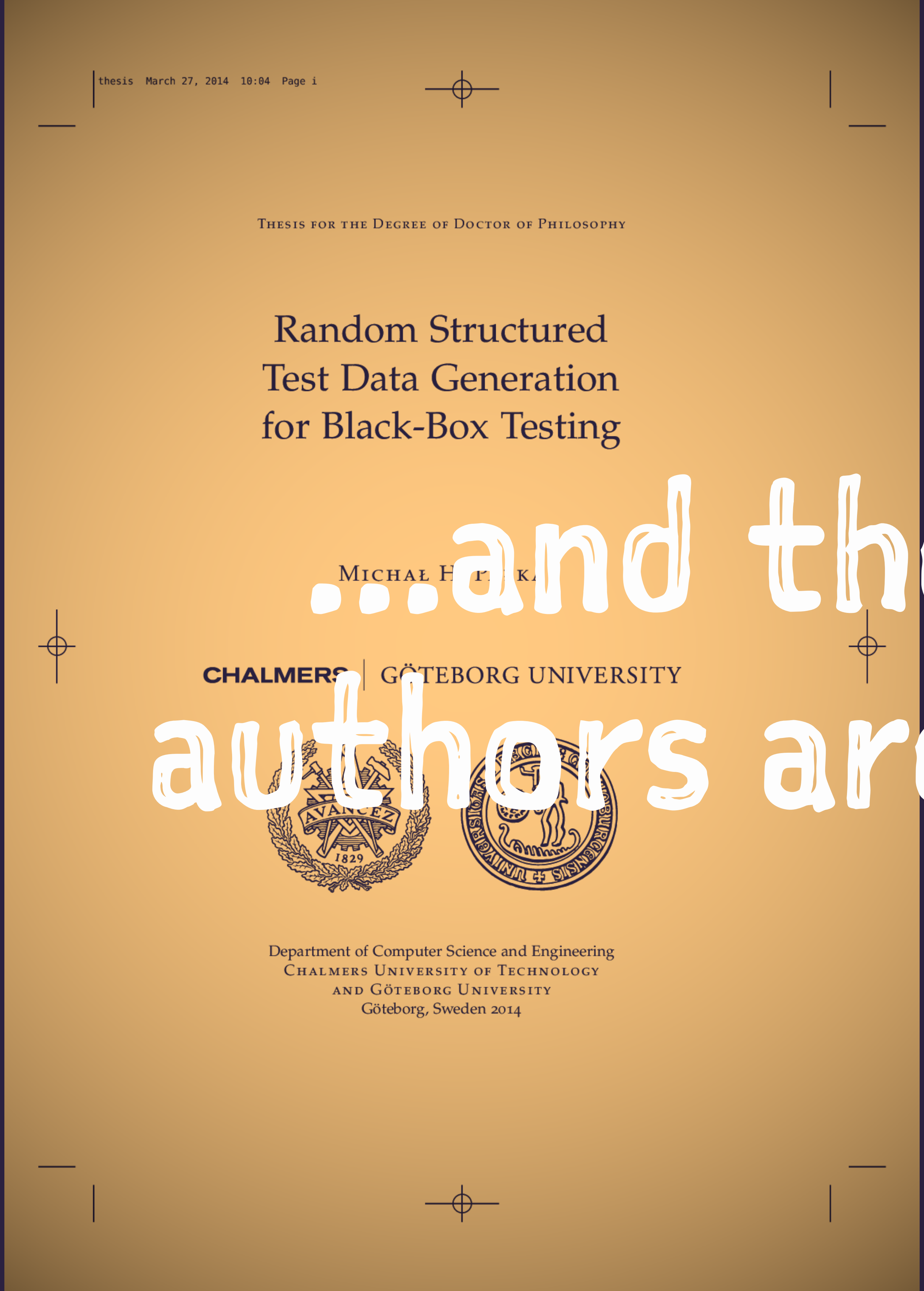
Koen made QuickCheck,

so... not surprised?

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 la... of the predicate, allowing us to pr... the space of values ...ly. In contrast, implementing test data generators by hand is ... 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.

1 Introduction

Random property-based testing has proven to be an effective method for finding bugs

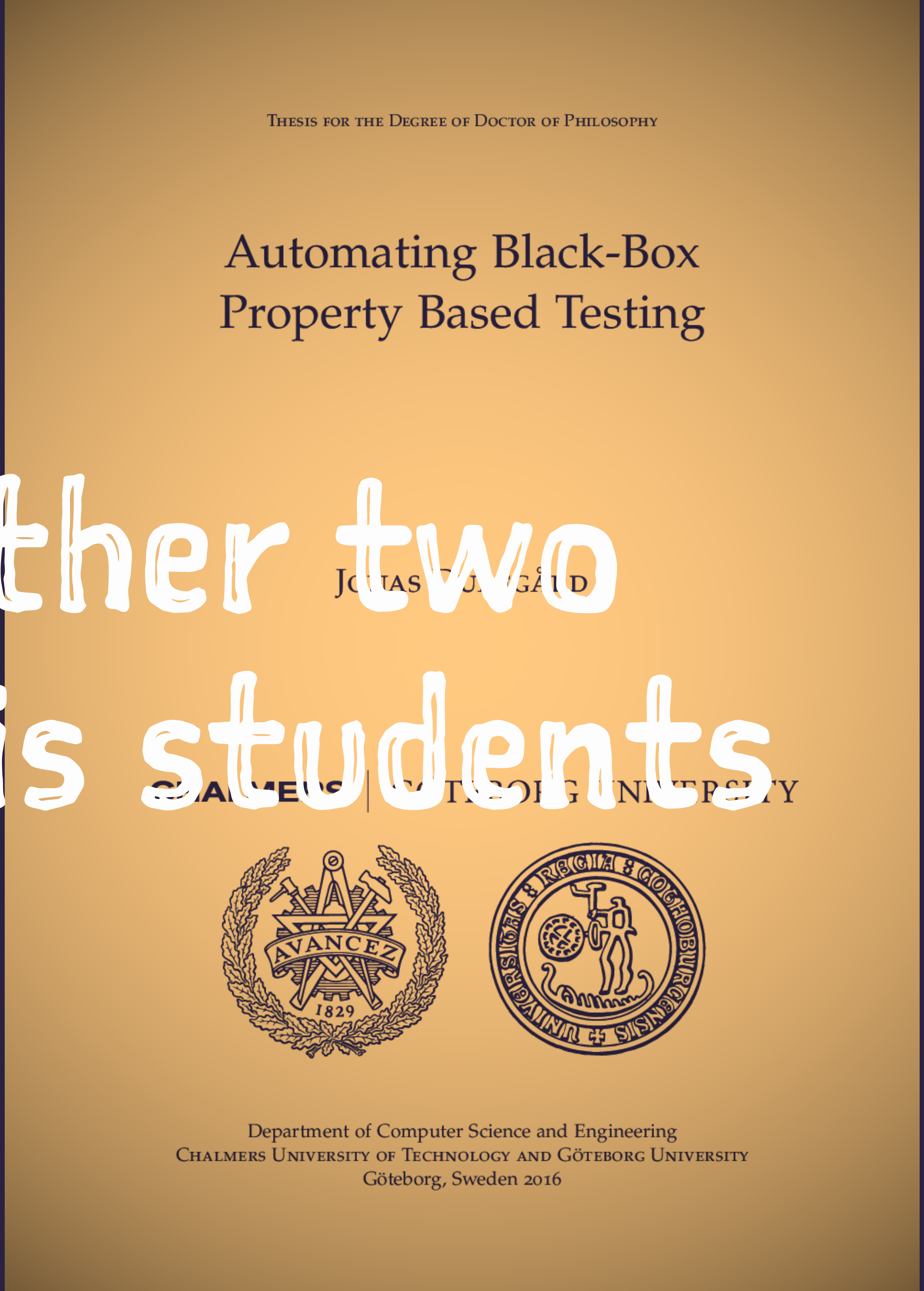


thesis March 27, 2014 10:04 Page i

Thesis for the Degree of Doctor of Philosophy

Random Structured Test Data Generation for Black-Box Testing

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

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

...and the other two authors are his students

So what is their trick?

Bag of tricks

1. do some really gross stuff to ensure sharing (implemented in the `size-based` package)
2. do some DSL magic to enumerate data types (implemented in the `testing-feat` package)
3. do some gross stuff to filter ill-typed terms eagerly (implemented in the `lazy-search` package)

Really gross stuff to ensure sharing

(hint: it's encapsulated global state)

DSL magic to enumerate data types

```
instance Enumerable Z where
    enumerate = datatype [] -- no longer a lie

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
7964948391145

-- how many programs of size <100?
$ eleanor --system Untyped --action Count --size 100
4503787720194931500936021688288566428450647198899831131920
```

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
38424355480696688647214814359468956803017461383159776132586

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

Gross stuff to filter ill-typed lamdas 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 lamdas 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!

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

(i.e. variables must be used exactly once)

This is affine!

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

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

This is neither!

```
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
```

(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?

What proportion of affine programs is linear?

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.

What proportion of affine programs is linear?

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

— xoxo Grygiel, Idiziak, and Zaionc

Ok sad 

(The chance of getting a linear program goes to zero as we increase program size... and pretty rapidly, actually...)

What can we do?

"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)

Mission: make check check linearity

```
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
```

Mission: make check check linearity

In a way which is parallelizable?

Mission: make check check linearity

In a way which is parallelizable?

Idea: when checking an application, try every possible split of variables between function and argument?

Uh, that sounds expensive?

I tried, it was

```
check :: Fin n => [(n, Type)] -> Type -> Term n -> Bool
check env a           (Var x)     = env == [(x, a)]
check env (a :-> b) (Lam t)     = check ((FZ, a) : map (first FS) env) b t
check env b           (App f s a) = or
    [ check env1 (a :-> b) f && check env2 a s
    | n <- [0..length env] , (env1, env2) <- combinations n env ]
check _   _           _           = False

combinations :: Int -> [a] -> [([a], [a])]
combinations 0 xs = [([], xs)]
combinations n (x:xs) =
    [ (x:xs, ys) | (xs, ys) <- combinations (n - 1) xs ] ++
    [ (xs, x:ys) | (xs, ys) <- combinations (n - 1) xs ]

-- how many linear lambdas of type Void :-> Void and size <30?
$ eleanor --system Linear --strategy Stupid --action Count --size 30
9790 -- took like a few hours
```

Mission: make check check linearity

In a way which is parallelizable? ❌

In a way which is eager?

Idea: use some state to keep track of whether a variable has been used yet?

Is that eager? 🤔

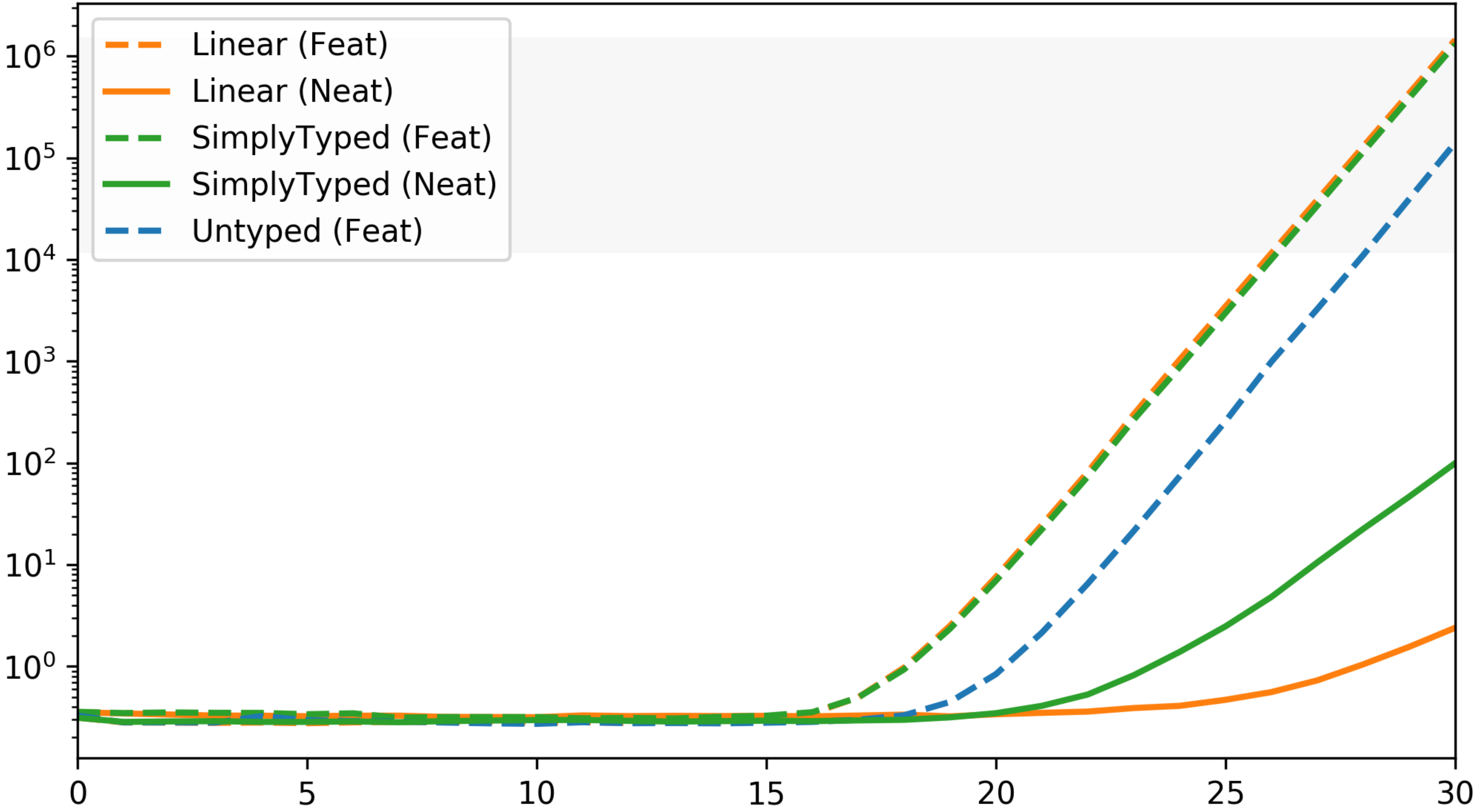
I tried

```
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
```

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

Future work

Can we make some BCI terms and translate them?

- it's way easier, but not complete 😕
- e.g. we don't get $\lambda x.\,(\lambda y.\,y)\;x$
- is that a problem?

Future work

Can we use the structure of linear programs to prune our search space?

- each term has n lambas, n vars, and $n-1$ apps
- kinda hard, probably won't scale well

Current work

Oh, right, I was testing a Rust library!

Let's see if that works now...

Oh no!

I'm out of time!

What have we seen?

When you write a compiler or library...

- think about what your thing does
- reference and actual implementation
- do they do the same stuff???

What have we seen?

When you wanna QuickCheck your compiler or library...

- generating random programs is really hard!
- but cool libraries have your back!
- even for newfangled linear and affine stuff

🤩