A TALE of TWO PROVERS

by Niki Vazou, Leonidas Lampropoulos and Jeff Polakow

Haskell

```
take :: Int → [a] → [a]
```

Liquid Haskell

```
take :: i:{Int|0≤i}→xs:{[a]|i≤len xs}→[a]
```

Liquid Haskell

```
take :: i:{Int | 0≤i} → xs:{[a] | i≤len xs} → [a]
```

```
take 2 [1,2,3]
```

OK

```
take 9 [1,2,3]
```

Error

Liquid Haskell

```
take :: i:{Int | 0≤i} → xs:{[a] | i≤len xs} → [a]
```

```
take 2 [1,2,3]
```

$0 \le 2 \le 3$

OK

SMT

```
take 9 [1,2,3]
```

$0 \le 9 \le 3$

Error

Is Liquid Haskell a Theorem Prover?

Is Liquid Haskell a Theorem Prover?

Theorem: *Parallelism Equivalence*
If `f` is a morphism*between two lists,
then `f` can be applied in parallel.

*`f` is a morphism when

```
f []=[] ∧ f (x<>y) = f x <> f y
```

Is Liquid Haskell a Theorem Prover?

Theorem: *Parallelism Equivalence*

If `f` is a morphism*between two lists,

then `f x = concat (pmap f (chunk i x))`.

*`f` is a morphism when

`f []=[] ∧ f (x<>y) = f x <> f y`

Is Liquid Haskell a Theorem Prover?

```
pEquiv :: f:([a] -> [b])
       -> Morphism [a] [b] f
       -> x:[a] -> i:Pos
       -> {f x = concat (pmap f (chunk i x))}
```

*f is a morphism when

```
f []=[] ∧ f (x<>y) = f x <> f y
```

Is Liquid Haskell a Theorem Prover?

```
pEquiv :: f:([a] -> [b])
       -> Morphism [a] [b] f
       -> x:[a] -> i:Pos
       -> {f x = concat (pmap f (chunk i x))}
```

```
*type Morphism a b f = x:a -> y:b ->
   {f []=[] ∧ f (x<>y) = f x <> f y}
```

Is Liquid Haskell a Theorem Prover?

Yes!

```
pEquiv :: f:([a] -> [b])
       -> Morphism [a] [b] f
       -> x:[a] -> i:Pos
       -> {f x = concat (pmap f (chunk i x))}
```

Theorems: Refinement Types

Proofs: (Terminating) Haskell Terms

Correctness: Liquid Type Checking

Is Liquid Haskell a Theorem Prover?

Yes!

```
pEquiv :: f:([a] -> [b])
       -> Morphism [a] [b] f
       -> x:[a] -> i:Pos
       -> {f x = concat (pmap f (chunk i x))}
```

Demo

Morphism Parallelism Equivalence

```
pEquiv :: (Chunkable n, Monoid m)
       => f:(n -> m)
       -> Morphism n m f
       -> x:n -> i:Pos
       -> {f x = mconcat (pmap f (chunk i x))}
```

Application: String Matching

Application: String Matching

Find all the occurrences of a *target* string in an *input* string.

Application: String Matching

Find all the occurrences of a *target* string in an *input* string.

“`the best of times`”

Application: String Matching

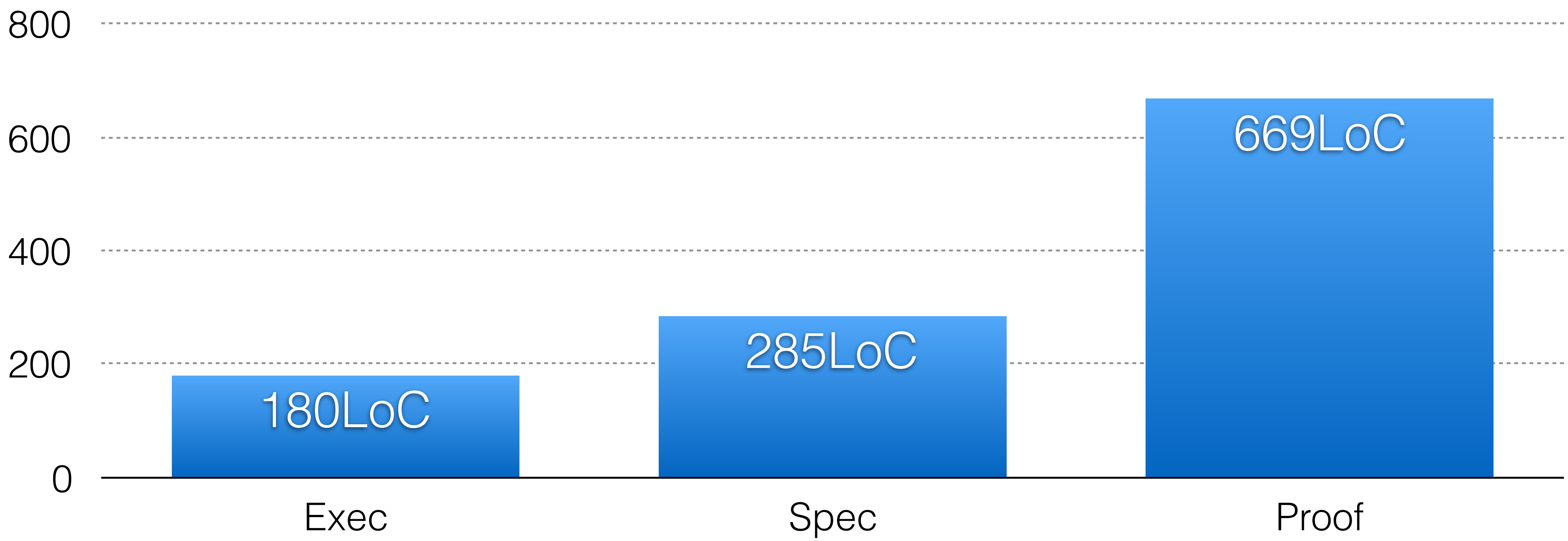
Find all the occurrences of a *target* string in an *input* string.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t	h	e		b	e	s	t		o	f		t	i	m	e	s

Target `“es”` matches at `[6, 16]`.

Application: String Matching

AWAKE

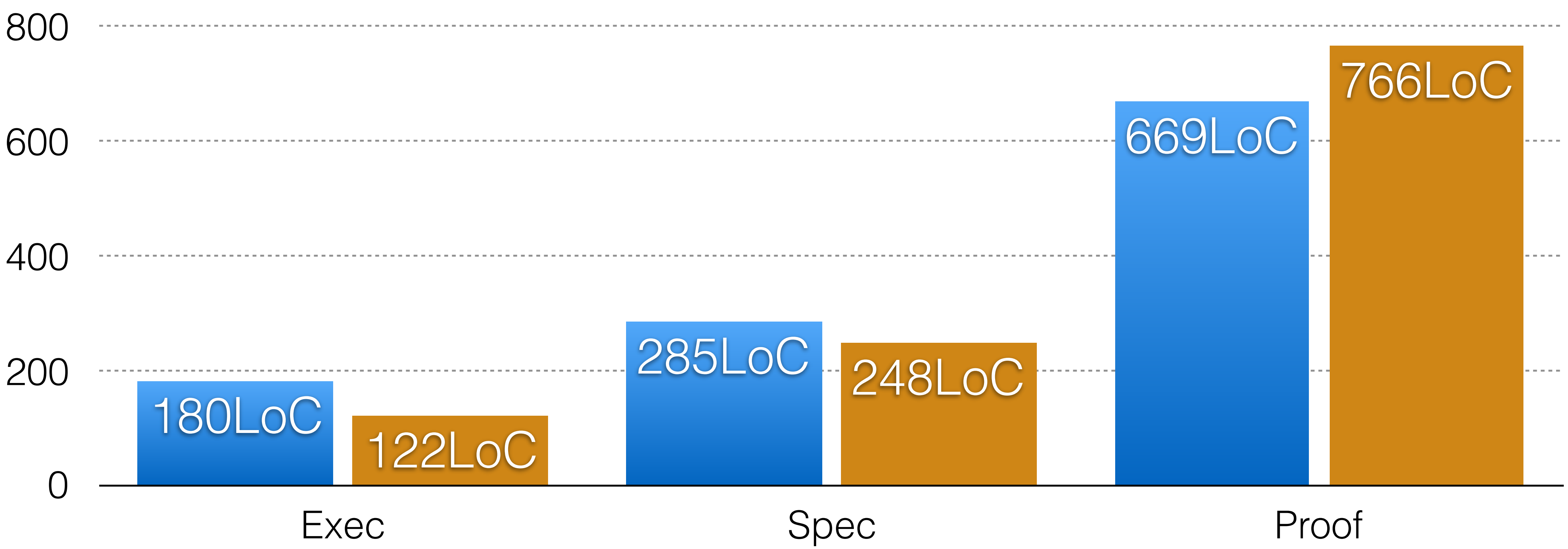


LoC (Proofs/Exec): 5x

Verification Time: 20 min

Human Effort: 2 months

vs.



LoC (Proofs/Exec):	5x	8x
Verification Time:	20 min	38 sec
Human Effort:	2 months	2 weeks

vs.

Haskell vs. **Non-Haskell** Proofs

VS.

Haskell VS. **Non-Haskell** Proofs

SMT- VS. **Tactic-** Based Automations

VS.

Haskell VS. **Non-Haskell** Proofs

SMT- VS. **Tactic-** Based Automations

Intrinsic VS. **Extrinsic** Verification

Intrinsic VS. **Extrinsic** Verification

```
take :: i:Nat → xs:{i≤len xs} → {v|len v=i}

take 0 _  = []
take i xs = x:take (i-1) xs
```

Intrinsic VS. **Extrinsic** Verification

```
take :: i:Nat → xs:{i≤len xs} → {v|len v=i}

take 0 _  = []
take i xs = x:take (i-1) xs
```

```
Definition take := seq.take.

Theorem take_spec:
 ∀i x, i≤length x→length (take i x)=i.
```

vs.

Haskell vs. **Non-Haskell** Proofs

SMT- vs. **Tactic-** Based Automations

Intrinsic vs. **Extrinsic** Verification

VS.

Haskell VS. **Non-Haskell** Proofs

SMT- VS. **Tactic-** Based Automations

Intrinsic VS. **Extrinsic** Verification

Semantic VS. **Syntactic** Termination

Semantic VS. **Syntactic** Termination

```
chunk :: i:Pos → xs:[a] → [[a]] / [len xs]
```

Semantic VS. Syntactic Termination

```
chunk :: i:Pos → xs:[a] → [[a]] / [len xs]
```

```
Fixpoint chunk {M: Type} (fuel: nat)
  (i: nat) (x: M) : option (list M)
```

Big VS. **Tiny** Trusted Code Base

```
chunk :: i:Pos → xs:[a] → [[a]] / [len xs]
```

ghc → → SMT → **OK** / **Error**

Big VS. **Tiny** Trusted Code Base

.hs → **ghc** → → **SMT** → **OK / Error**

VS.

Haskell VS. **Non-Haskell** Proofs

SMT- VS. **Tactic-** Based Automations

Intrinsic VS. **Extrinsic** Verification

Semantic VS. **Syntactic** Termination

Big VS. **Tiny** Trusted Code Base

VS.

Haskell VS. **Non-Haskell** Proofs

SMT- VS. **Tactic-** Based Automations

Intrinsic VS. **Extrinsic** Verification

Semantic VS. **Syntactic** Termination

Big VS. **Tiny** Trusted Code Base

Proof **Verifier** VS. **Assistant**

Conclusion

Liquid Haskell *is a promising prover, but needs a lot of* ***Coq****-inspired future work.*

Conclusion

Liquid Haskell *is a promising prover, but needs a lot of* ***Coq****-inspired future work.*

Fast "tactics"

Liquid GUI Proof Assistant

Hackage Sharing Proofs

Thanks!