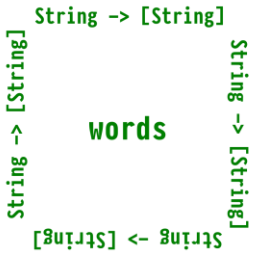
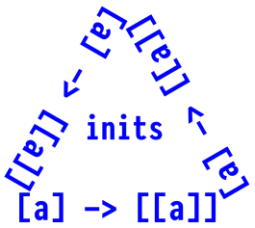
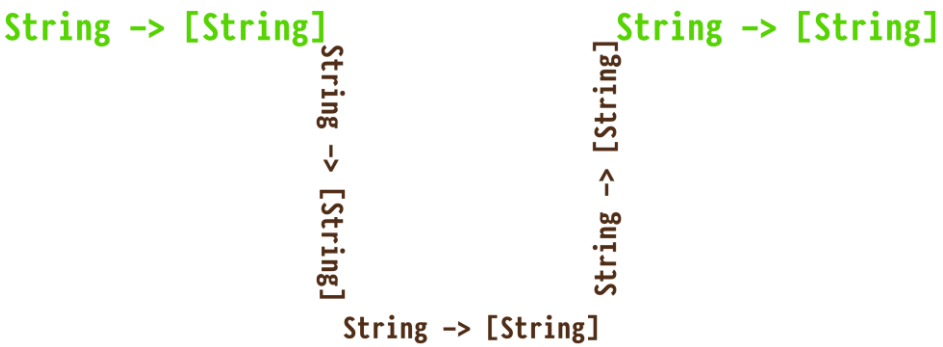
Haskell Symposium 2018

Suggesting Valid Hole Fits for Typed-Holes (Experience Report)







Matthías Páll Gissurarson

(Chalmers University of Technology)

CHALMERS UNIVERSITY OF TECHNOLOGY

OCTOPI

Typed-Holes

GHC 7.8.1

```
f :: [String]
f = _ "hello, world"
```

- Found hole: `_ :: [Char] -> [String]`
- In the expression: `_`

  In the expression: `_ "hello, world"`

  In an equation for ‘f’: `f = _ "hello, world"`
- Relevant bindings include

  `f :: [String]`(bound at t.hs:2:1)

Using with Lens

```haskell
import Control.Lens
import Control.Monad.State

newtype T = T { _v :: Int }

val :: Lens' T Int
val f (T i) = T <$> f i

updT :: T -> T
updT t = t &~ do
  val `_` (1 :: Int)
```

```
Found hole:
  _ :: ((Int -> f0 Int) -> T -> f0 T) -> Int -> State T a0
Where: ‘f0’ is an ambiguous type variable
       ‘a0’ is an ambiguous type variable
In the expression: _
  In a stmt of a 'do' block: val `_` (1 :: Int)
  In the second argument of ‘(&~)’, namely ‘do val `_` (1 :: Int)’
Relevant bindings include
  t :: T (bound at Lens.hs:11:6)
  updT :: T -> T (bound at Lens.hs:11:1)
```

Using with Lens

```
Found hole:
  _ :: ((Int -> f0 Int) -> T -> f0 T) -> Int -> State T a0
Valid hole fits include
  (#=)  :: MonadState s m => ALens s s a b -> b -> m ()
  (<#=) :: MonadState s m => ALens s s a b -> b -> m b
  (<*=) :: (MonadState s m, Num a) =>
           LensLike' ((,) a) s a -> a -> m a
  (<+=) :: (MonadState s m, Num a) =>
           LensLike' ((,) a) s a -> a -> m a
  (<-=) :: (MonadState s m, Num a) =>
           LensLike' ((,) a) s a -> a -> m a
  (<<*=) :: (MonadState s m, Num a) =>
            LensLike' ((,) a) s a -> a -> m a
  …
```

```
import Control.Lens
import Control.Monad.State

newtype T = T { _v :: Int }

val :: Lens' T Int
val f (T i) = T <$> f i

updT :: T -> T
updT t = t &~ do
  val `_` (1 :: Int)
```

Valid Hole Fits

GHC 8.4.1

```
f :: [String]
f = _ "hello, world"

Valid hole fits include
  lines :: String -> [String]
  words :: String -> [String]
  repeat :: a -> [a]
  fail :: Monad m => String -> m a
  return :: Monad m => a -> m a
  pure :: Applicative f => a -> f a
  …
```

Refinement Hole Fits

GHC 8.6.1

```
f :: [String]
f = _ "hello, world"

Valid refinement hole fits include
  iterate (_ :: String -> String)
    where iterate :: (a -> a) -> a -> [a]
  replicate (_ :: Int)
    where replicate :: Int -> a -> [a]
  …
  map (_ :: Char -> String)
    where map :: (a -> b) -> [a] -> [b]
  fmap (_ :: Char -> String)
    where fmap :: Functor f => (a -> b) -> f a -> f b
  …
```

Demo

Using with Non-Functional Properties

```
> _ [3,1,2] :: Sorted (O(N*.LogN)) (O(N)) Integer
```

- Found hole:

```
_ :: [Integer] -> Sorted (O ('NLogN 1 1)) (O N) Integer

…

Valid hole fits include
  mergeSort :: forall (n :: AsymP) (m :: AsymP) a.
                 (n >=. O (N *. LogN), m >=. O N, Ord a) =>
                 [a] -> Sorted n m a
  quickSort :: forall (n :: AsymP) (m :: AsymP) a.
                 (n >=. O (N *. LogN), m >=. O LogN, Ord a) =>
                 [a] -> Sorted n m a
```

Using with the Free Monad

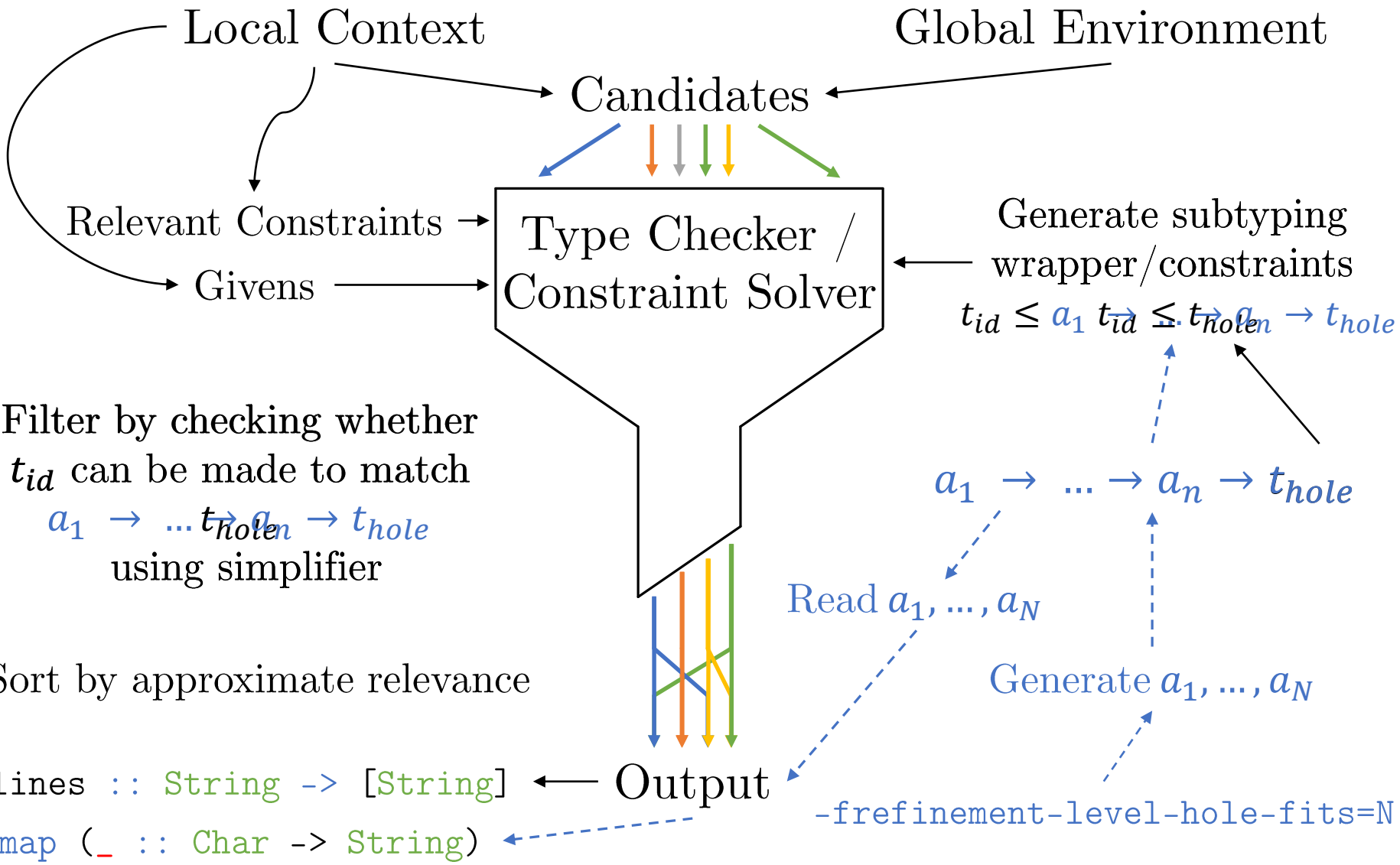
```
Found hole: _ :: Free f a -> Free f b

…

Valid refinement hole fits include
  fmap (_ :: a -> b)
    where fmap :: Functor f => (a -> b) -> f a -> f b
  (<*>) (_ :: Free f (a -> b))
    where (<*>) :: Applicative f => f (a -> b) -> f a -> f b
  (<$>) (_ :: a -> b)
    where (<$>) :: Functor f => (a -> b) -> f a -> f b
  (=<<) (_ :: a -> Free f b)
    where (=<<) :: Monad m => (a -> m b) -> m a -> m b
  (<*) (_ :: Free f b)
    where (<*) :: Applicative f => f a -> f b -> f a
  ($) (_ :: b)
    where ($) :: Functor f => a -> f b -> f a
  …
```

```
data Free f a = Pure a | Free (f (Free f a))
…
instance Functor f => Monad (Free f) where
    return a      = Pure a
    Pure a >>= f  = f a
    Free f >>= g  = Free (fmap _ f)
```

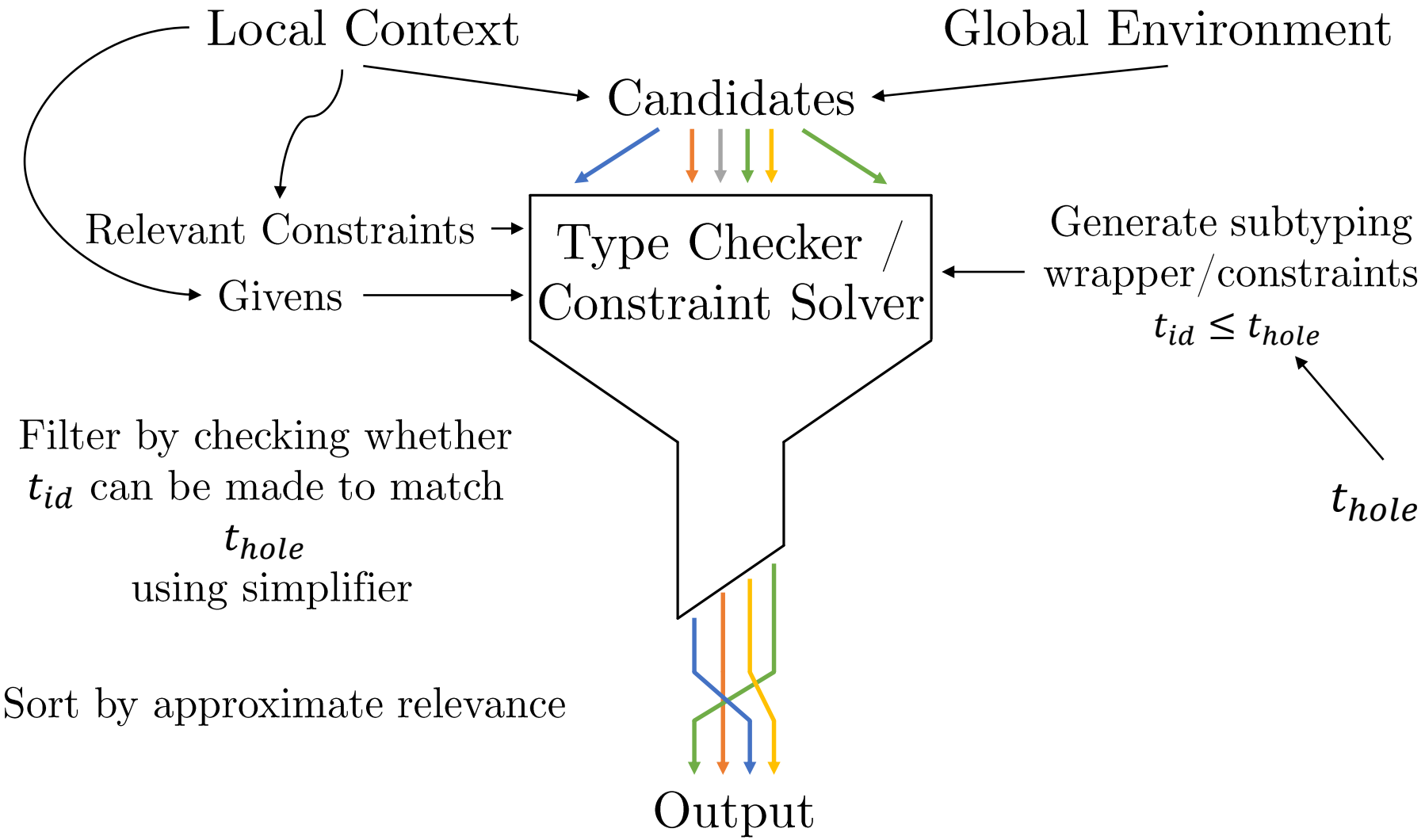
Implementation

Checking for Hole Fits (on `HoleError`)

Refinement

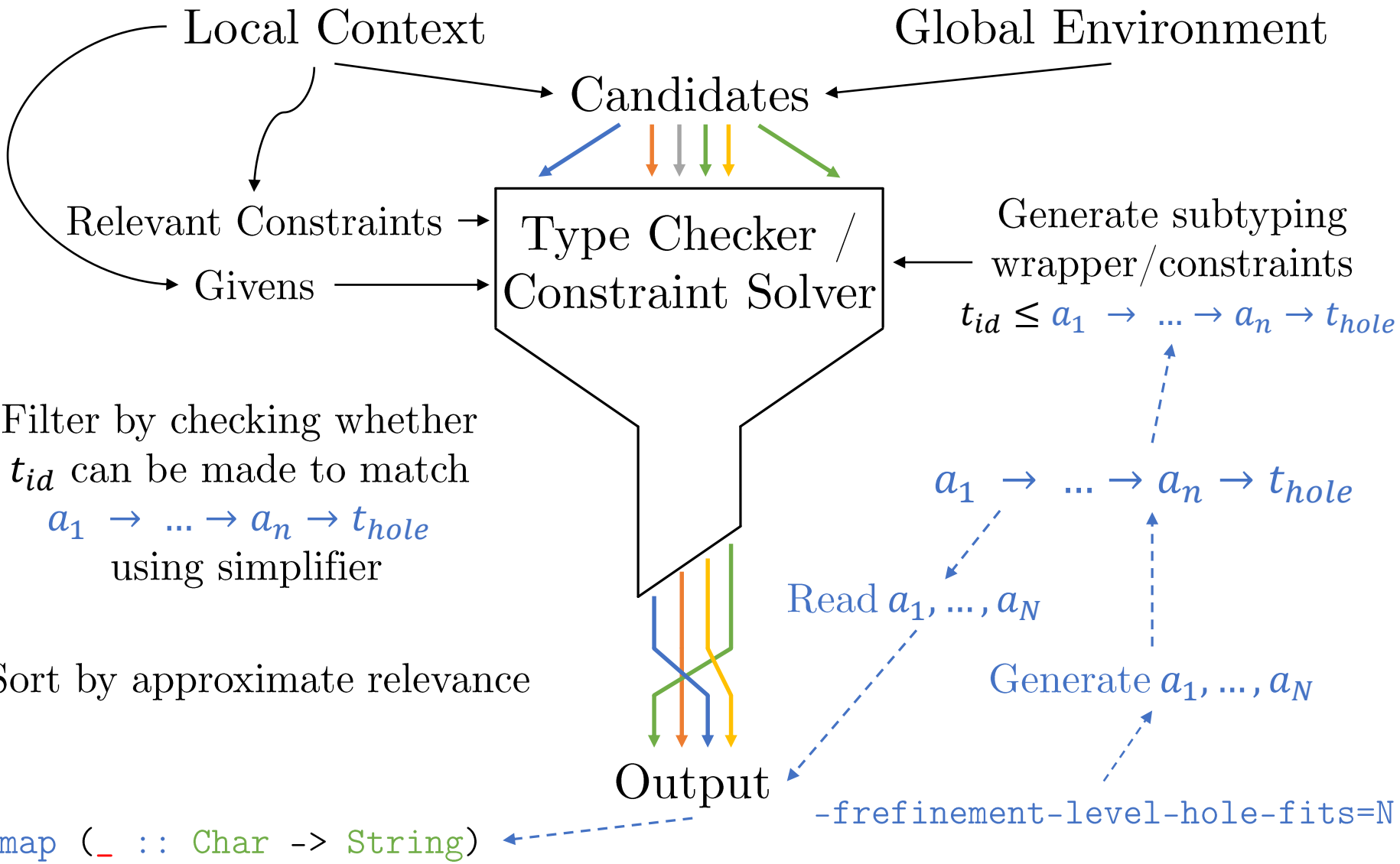


Checking for Hole Fits (on `HoleError`)



Checking for Hole Fits (on `HoleError`)

Refinement



Sorting the Output

```
f :: [String]
f = _ "hello, world"

Valid hole fits include
  lines :: String -> [String]
  words :: String -> [String]
  repeat :: a -> [a]
  fail :: Monad m => String -> m a
  return :: Monad m => a -> m a
  pure :: Applicative f => a -> f a
  (Some hole fits suppressed; use -fmax-valid-hole-fits=N
   or -fno-max-valid-hole-fits)
```

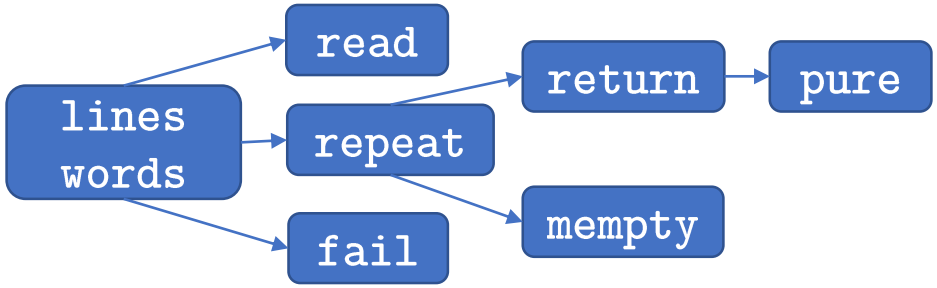
Sorting by Size (of the type application)

- For `_ :: String -> [String]`

Hole Fit	Type	Application	Size
`lines`	`String -> [String]`		0
`repeat`	`a -> [a]`	`String`	2
`mempty`	`Monoid a => a`	`String -> [String]`	6

`String = [Char]` ⇒ 2 constructors

Sorting by Subsumption



Subsumption graph for `_ :: String -> [String]`

```
lines :: String -> [String]
words :: String -> [String]
read :: Read a => String -> a
repeat :: a -> [a]
mempty :: Monoid a => a
return :: Monad m => a -> m a
pure :: Applicative f => a -> f a
fail :: Monad m => String -> m a
```

Subsumption

```
lines :: String -> [String]
words :: String -> [String]
repeat :: a -> [a]
fail :: Monad m => String -> m a
return :: Monad m => a -> m a
pure :: Applicative f => a -> f a
read :: Read a => String -> a
mempty :: Monoid a => a
```

By Size

`-fsort-by-subsumption-hole-fits`

Conclusion

- Allows users to search type-level documentation directly.
- Facilitates type-driven development (TDD).
- Lightweight (uses existing machinery).
- Non-intrusive (contained in one module).
- Available now!

Future Work

- Considering built-in syntax such as `(:)` or `[]`
- Showing more specific fits, e.g.

  `pi :: Floating a => a` for `_ :: Fractional a => a`
- Functions with fewer arguments (or in a different order).
- Allowing users to specify invariants to filter by behavior.

```
{-@ isPositive :: x:Int -> {v:Bool | v <=> x > 0} @-}
```

(Liquid Haskell)

Thank You!

dl.acm.org/citation.cfm?doid=3242744.3242760

||

mpg.is/papers/gissurarson2018suggesting.pdf

Bonus Slides

Using with Lens

`-fshow-docs-of-hole-fits`

```
Found hole:
  _ :: ((Int -> f0 Int) -> T -> f0 T) -> Int -> State T a0
Valid hole fits include
  (#=) :: MonadState s m => ALens s s a b -> b -> m ()
  (<#=) :: MonadState s m => ALens s s a b -> b -> m b
  (<*=) :: (MonadState s m, Num a) =>
           LensLike' ((,) a) s a -> a -> m a
  (<+=) :: (MonadState s m, Num a) =>
           LensLike' ((,) a) s a -> a -> m a
  (<-=) :: (MonadState s m, Num a) =>
           LensLike' ((,) a) s a -> a -> m a
  (<<*=) :: (MonadState s m, Num a) =>
            LensLike' ((,) a) s a -> a -> m a
  …
```

```haskell
import Control.Lens
import Control.Monad.State

newtype T = T { _v :: Int }

val :: Lens' T Int
val f (T i) = T <$> f i

updT :: T -> T
updT t = t &~ do
    val `_` (1 :: Int)
```

Using with Lens

On master

-fshow-docs-of-hole-fits

```
Found hole:
  _ :: ((Int -> f0 Int) -> T -> f0 T) -> Int -> State T a0
Valid hole fits include
  (#=) :: MonadState s m => ALens s s a b -> b -> m ()
    {-^ A version of ('Control.Lens.Setter..=') that works on 'ALens'.-}
  (<#=) :: MonadState s m => ALens s s a b -> b -> m b
    {-^ A version of ('Control.Lens.Setter.<.=') that works on 'ALens'.-}
  (<*=) :: (MonadState s m, Num a) =>
           LensLike' ((,) a) s a -> a -> m a
    {-^ Multiply the target of a numerically valued 'Lens' into your 'Monad''s
        state and return the result. …
  (<+=) :: (MonadState s m, Num a) =>
           LensLike' ((,) a) s a -> a -> m a
    {-^ Add to the target of a numerically valued 'Lens' into your 'Monad''s state
        and return the result. …
  …
```