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

Matthías Páll Gissurarson
Chalmers University of Technology, Sweden
pallm@chalmers.se

Abstract

Type systems allow programmers to communicate a partial specification of their program to the compiler using types, which can then be used to check that the implementation matches the specification. But can the types be used to aid programmers during development? In this experience report I describe the design and implementation of my lightweight and practical extension to the typed-holes of GHC that improves user experience by adding a list of *valid hole fits* and *refinement hole fits* to the error message of typed-holes. By leveraging the type checker, these fits are selected from identifiers in scope such that if the hole is substituted with a valid hole fit, the resulting expression is guaranteed to type check.

CCS Concepts • **Software and its engineering** → *Compilers*; *Software development techniques*;

Keywords GHC, Typed-Holes, Valid Hole Fits, Suggestions

ACM Reference Format:
Matthías Páll Gissurarson. 2018. Suggesting Valid Hole Fits for Typed-Holes (Experience Report). In *Proceedings of the 11th ACM SIGPLAN International Haskell Symposium (Haskell '18), September 27-28, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3242744.3242760

1 Introduction

When writing documentation for libraries, the Haskell community often goes the route of having descriptive function names and clear types that leverage type synonyms in order to push much of the documentation to the type-level. As developers program in Haskell, they often use a style of programming called *Type-Driven Development*. They write out the input and output types of functions before writing the functions themselves [3]. A consequence of this approach is that the compiler has a lot of type information that is only used during type checking. Can we make better use of the extra information and type-level documentation and improve user experience? According to the GitHub survey [5], user experience is the third most important factor when choosing open source software, after stability and security, and thus an important consideration.

We can leverage the richness of type information in library documentation along with users' type signatures by extending typed-hole error messages with a list of *valid hole fits* and *refinement hole fits*. These allow users to find relevant functions and constants when a typed-hole is encountered:

```
Found hole: _ :: [Int] -> Int
In the expression: _ :: [Int] -> Int
In an equation for ‘it’: it = _ :: [Int] -> Int
Relevant bindings include
  it :: [Int] -> Int (bound at <interactive>:4:1)
Valid hole fits include
  head :: forall a. [a] -> a
  last :: forall a. [a] -> a
  length :: forall (t :: * -> *) a. Foldable t => t a -> Int
  maximum :: forall (t :: * -> *) a.
             (Foldable t, Ord a) => t a -> a
  minimum :: forall (t :: * -> *) a.
             (Foldable t, Ord a) => t a -> a
  product :: forall (t :: * -> *) a.
             (Foldable t, Num a) => t a -> a
  (Some hole fits suppressed; use
   -fmax-valid-hole-fits=N or -fno-max-valid-hole-fits)
Valid refinement hole fits include
  foldl1 (_ :: Int -> Int -> Int)
    where foldl1 :: forall (t :: * -> *) a. Foldable t =>
                    (a -> a -> a) -> t a -> a
  foldr1 (_ :: Int -> Int -> Int)
    where foldr1 :: forall (t :: * -> *) a. Foldable t =>
                    (a -> a -> a) -> t a -> a
  foldl (_ :: Int -> Int -> Int) (_ :: Int)
    where foldl :: forall (t :: * -> *) b a. Foldable t =>
                   (b -> a -> b) -> b -> t a -> b
  foldr (_ :: Int -> Int -> Int) (_ :: Int)
    where foldr :: forall (t :: * -> *) a b. Foldable t =>
                   (a -> b -> b) -> b -> t a -> b
  ($) (_ :: [Int] -> Int)
    where ($) :: forall a b. (a -> b) -> a -> b
  const (_ :: Int)
    where const :: forall a b. a -> b -> a
  (Some refinement hole fits suppressed;
   use -fmax-refinement-hole-fits=N
   or -fno-max-refinement-hole-fits)
```

Figure 1. Typed-hole error message extended with hole fits.

Haskell '18, September 27-28, 2018, St. Louis, MO, USA

ACM ISBN 978-1-4503-5835-4/18/09…$15.00
https://doi.org/10.1145/3242744.3242760

Valid hole fits and refinement hole fits can be used to effectively aid development in many scenarios by allowing users to view and search type-level documentation directly, thus improving the user experience.

Note: in the interest of reducing noise in the output in this report, I have opted to show only the fits themselves, and not the type application nor provenance of the fit as displayed in the ouput by default. The amount of detail in the output is controlled by flags; the format used here is achieved by setting the `-funclutter-valid-hole-fits` flag. An example of the full default output can be seen in figure 2.

```
product :: forall (t :: * -> *) a.
           (Foldable t, Num a) => t a -> a
  with product @[] @Int
  (imported from ‘Prelude’
   (and originally defined in ‘Data.Foldable’))
```

Figure 2. The full output for a fit for `_ :: [Int] -> Int`.

1.1 Contributions

In this experience report, I do the following:

- Describe *valid hole fits* and *refinement hole fits* as I have implemented them in GHC. Valid hole fits allow users to tap in to the extra type information available during compilation or interactively using GHCi, while refinement hole fits extend valid hole fits beyond identifiers to find functions that need additional arguments.
- Provide a detailed explanation of how I have implemented valid hole fits and refinement hole fits in GHC, and how I solved technical hurdles along the way.
- Show the usefulness of hole fits in case studies on an introductory exercise and when using the `lens` library.
- Finally, I present an application of valid hole fits to libraries using type-in-type to annotate functions with non-functional properties, and show an example.

1.2 Background

Typed-Holes in GHC were introduced in version *7.8* and implemented by Simon Peyton Jones, Sean Leather and Thijs Alkemade [7]. Inspired by a similar feature in Agda, typed-holes allow a user of GHC to have "holes" in their code, using an underscore (`_`) in place of an expression. When GHC encounters a typed-hole, it generates an error with information about that hole, such as its location, the (possibly inferred) type of the hole and relevant local bindings [4]. Typed-holes can also be given names by appending characters, e.g. `_a` and `_b`, to allow users to distinguish between holes.

Valid Hole Fits: We use the type information available in typed-holes to make them more useful for programmers, by extending the typed-hole error message with a list of *valid hole fits*. Valid hole fits are expressions which the hole can be replaced with directly, and the resulting expression will type check. An example of valid hole fits can be seen in figure 1.

Refinement Hole Fits: It is often the case that a single identifier is not enough to implement the desired function, such as when writing the `product` function (`foldr (*) 1`). To suggest useful hole fits for these cases, we introduce *refinement hole fits*. Refinement hole fits are valid hole fits that have one or more additional holes in them. The number of additional holes is controlled by the refinement level, set via `-frefinement-level-hole-fits`. A refinement level of N means that hole fits with up to N additional holes in them will be considered. An example of refinement hole fits can be seen in figure 1, in which the refinement level is 2.

2 Case Studies

To show that valid hole fits and refinement hole fits can be used to effectively aid development, we consider two cases, an introductory programming exercise where we use the `Prelude` and an advanced case using the `lens` library.

2.1 Exercise from Programming in Haskell

To study how the valid hole fits perform when used by beginners, I looked at an example from Graham Hutton's introductory text, Programming in Haskell [9]. In exercise 4.8.1, students are asked to implement `halve :: [a] -> ([a],[a])`, which should split a list of even length into two halves. With refinement hole fits enabled, we can query GHCi by writing:

```
Prelude> _ :: [a] -> ([a], [a])
```

In response, GHCi will then generate a typed-hole error, including a list of valid refinement hole fits:

```
Valid refinement hole fits include
  break (_ :: a1 -> Bool)
    where break :: forall a.
                   (a -> Bool) -> [a] -> ([a], [a])
  span (_ :: a1 -> Bool)
    where span :: forall a.
                  (a -> Bool) -> [a] -> ([a], [a])
  splitAt (_ :: Int)
    where splitAt :: forall a. Int -> [a] -> ([a], [a])
  mapM (_ :: a1 -> ([a1], a1))
    where mapM :: forall (t :: * -> *) (m :: * -> *) a b.
                  (Traversable t, Monad m) =>
                  (a -> m b) -> t a -> m (t b)
  traverse (_ :: a1 -> ([a1], a1))
    where traverse :: forall (t :: * -> *) (f :: * -> *) a b.
                      (Traversable t, Applicative f) =>
                      (a -> f b) -> t a -> f (t b)
  const (_ :: ([a1], [a1]))
    where const :: forall a b. a -> b -> a
  (Some refinement hole fits suppressed;
   use -fmax-refinement-hole-fits=N
   or -fno-max-refinement-hole-fits)
```

One of the suggested fits is the `splitAt (_ :: Int)` refinement, and given that the task is to *split* a list, this seems like a good fit. In this way, the student can discover the `splitAt` function from the prelude, and a correct solution (`halve xs = splitAt (length xs `div` 2) xs`) is easy to find using refinement hole fits.

2.2 The Lens Library

In the `lens` library [10], the functions can be hard to find with Hoogle (see section 5), due to the library's extensive use of type synonyms. As an example, consider the following:

```
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)
```

For the hole in the above, the typed-hole message includes:

```
Found hole:
  _ :: ((Int -> f0 Int) -> T -> f0 T) -> Int -> State T a0
```

where `f0` and `a0` are ambiguous type variables. Searching for this type signature in Hoogle (version 5.0.17) yields no results from the lens library.

When valid hole fits are available, GHC will output the following list of valid hole fits:

```
Valid hole fits include
  (#=) :: forall s (m :: * -> *) a b. MonadState s m =>
          ALens s s a b -> b -> m ()
  (<#=) :: forall s (m :: * -> *) a b. MonadState s m =>
           ALens s s a b -> b -> m b
  (<*=) :: forall s (m :: * -> *) a. (MonadState s m,
           Num a) => LensLike' ((,) a) s a -> a -> m a
  (<+=) :: forall s (m :: * -> *) a. (MonadState s m,
           Num a) => LensLike' ((,) a) s a -> a -> m a
  (<-=) :: forall s (m :: * -> *) a. (MonadState s m,
           Num a) => LensLike' ((,) a) s a -> a -> m a
  (<<*=) :: forall s (m :: * -> *) a. (MonadState s m,
            Num a) => LensLike' ((,) a) s a -> a -> m a
  (Some refinement hole fits suppressed;
   use -fmax-refinement-hole-fits=N
   or -fno-max-refinement-hole-fits)
```

Though the names of the functions are opaque, we see that integrating the valid hole fits into the typed-holes and integrating with the type checker itself is a clear win, allowing us to find a mulitude of relevant functions from `lens`.

3 Implementation

The valid hole fit suggestions for typed-holes are implemented as an extension to the error reporting mechanism of GHC, and are only generated during error reporting of holes. This means that we can emphasize utility rather than performance, as any overhead will only be incurred when the program would in any case fail due to an error.

3.1 Inputs & Outputs

The entry into the valid hole fit search is the function called `findValidHoleFits` in the **`TcHoleErrors`** module [1]:

```
findValidHoleFits :: TidyEnv -- Type env for zonking
                  -> [Implication] -- Enclosing implics
                                   -- containing givens
                  -> [Ct] -- Unsolved simple constraints
                          -- in the implic for the hole.
                  -> Ct -- The hole constraint itself
                  -> TcM (TidyEnv, SDoc)
```

This function takes the hole constraint that caused the error, the unsolved simple constraints that were in the same set of wanted constraints as the hole constraint, and the list of implications which that set was nested in. The tidy type environment at that point of error reporting is also passed to the function, and used later for *zonking* [2]. To zonk, we use

```
zonkTidyTcType :: TidyEnv -> TcType -> TcM (TidyEnv, TcType)
```

from **`TcMType`**, which uses the tidy type environment to ensure that the resulting types are consistent with the rest of the error message and other error messages. The function returns the (possibly) updated tidy type environment and the message containing the valid hole fits.

3.2 Relevant Constraints

The unsolved simple constraints are constraints imposed by the call-site of the hole. As an example, consider the holes `_a` and `_b` in the following:

```
f :: Show a => a -> String
f x = show (_b (show _a))
```

Here, the type of `_a` and the return type `_b` need to fulfill a show constraint. These constraints constitute the set of unsolved simple constraints {**`Show`** t_a, **`Show`** t_b}, where t_a is the type of `_a`, and **`String`** **`->`** t_b is the type of `_b`. Since valid hole fits are only considered for one hole at a time, the unsolved simple constraints are filtered to only contain constraints relevant to the current hole. For hole `_a`, this would be {**`Show`** t_a}, and for hole `_b` this would be {**`Show`** t_b}. This is done by discarding those constraints whose types do not share any free type variables with the type of the hole. I call this filtered set of constraints the *relevant constraints*.

3.3 Candidates

Candidate hole fits are identifiers gathered from the environment. We consider only the elements in the global reader and the local bindings at the location of the hole (discarding any shadowed bindings). The global reader contains identifiers that are imported or defined at the top-level of the module. Using the local bindings allows us to include candidates bound by pattern matching (such as function arguments) or in **`let`** or **`where`** clauses. As an example, in:

```
f (x:xs) = let a = () in _
  where k = head xs
```

the global reader elements considered as candidates are the functions in the **`Prelude`** and `f`, while the local binding candidates are `f`, `x`, `xs`, `a` and `k`. When shadowed bindings are

[1] Available in GHC HEAD at:
http://git.haskell.org/ghc.git/blob/refs/heads/master:/compiler/typecheck/TcHoleErrors.hs

[2] In the context of GHC, *zonking* is when a type is traversed and mutable type variables are replaced with the real types they dereference to.

removed, the `f` from the global reader is discarded. For global elements, a lookup is performed in the type checker to find their associated identifiers, discarding any elements not associated with an identifier or data constructor (like type constructors or type variables). Candidates from `GHC.Err` (like `undefined`) are discarded, since they can be made to match any type at all, and are unlikely to be the function that the user is looking for.

3.4 Checking for Fit

Each of these candidates is checked in turn by invoking the `tcCheckHoleFit` function. This function starts by capturing the set of constraints and wrapper emitted by the `tcSubType_NC` function when invoked on the type of the candidate and the type of the hole. The `tcSubType_NC` function takes in two types and returns the core wrapper needed to go from one type to the other, emitting the constraints which must be satisfied for the types to match. The relevant constraints are added to this set of constraints, to ensure that any constraints imposed by the call-site of the hole are satisfied as well. This extended set is wrapped in the implications that the hole was nested in, so that any givens contained in the implications (such as that `a` satisfies the show constraint in the example above) are passed along. These are passed to the simplifier, which checks the constraints. If the set is soluble, the candidate is a valid hole fit, and the wrapper is returned. The wrapper is used later to show *how* the type of the fit matches the type of the hole by showing the type application, like `product @[] @Int` in figure 2.

3.5 Refinement hole fits

For refinement hole fits, N fresh flexible type variables are created, $a_1, \ldots, a_N$, where N is the refinement level set by the `-frefinement-level-hole-fits` flag. We then look for fits not for the type of the hole, t_h, but for the type $a_1 \to \cdots \to a_N \to t_h$. These additional type variables allow us to emulate additional holes in the expression. To limit the number of refinement hole fits, additional steps are taken after we have checked whether the type fits, to check whether all the fresh type variables ended up being unified with a concrete type. This ensures that fits involving fresh variables such as `id (_ :: a1 -> a2 -> a) (_ :: a1) (_ :: a2)` are discarded unless explicitly requested by the user by passing the `-fabstract-refinement-hole-fits` flag. If a match is found, the fresh type variables are zonked and the type they were unified with read off them, allowing us to show the types of the additional holes (like `Int -> Int -> Int` for the hole in the `foldl1 (_ :: Int -> Int -> Int)` fit).

3.6 Sorting the Output

As with relevant bindings, only 6 valid hole fits are displayed by default. To increase the utility of the valid hole fits, we sort the fits by relevance, which is approximated in two ways.

Sorting by Size: The default approximation sorts by the size of unique types in the type application needed to go from the type of the fit to the type of the hole, as defined by the core expression wrapper returned when the fit was found. The size is computed by applying the `sizeTypes` function, which counts the number of variables and constructors:

Table 1. Sizes of matches for `_ :: String -> [String]`

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

Only unique types are considered, since fits that require many different types are in some sense "farther away" than fits that require only a few unique types. This method is faster and returns a reasonable ordering in most cases.

Sorting by Subsumption: The other approximation is enabled by the `-fsort-by-subsumption-hole-fits` flag. When sorting by subsumption, a subsumption graph is constructed by checking all the fits that have been found for whether they can be used in place of any other found fit. A directed graph is made, in which the nodes are fits and the edges are the result of the subsumption check, where fit a has an edge to fit b fit if b could be used anywhere that a could be used. An example of such a graph can be seen in figure 3. The fits are sorted by a topological sort on this graph, so that if b could be used anywhere a could be used, then b appears after a in the output. This ordering ensures that more specific fits (such as those with the same type as the hole) appear earlier than more abstract, general fits.

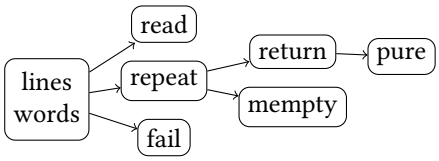

Figure 3. The subsumption graph for matches for `_ :: String -> [String]`. Here `lines` would come before `repeat`, `read`, and `fail`, `repeat` before `mempty` and `return`, etc.

3.7 Dealing with Side-effects

When GHC simplifies constraints, it does so by side-effect on the type variables involved and the evidence contained within implications. To ensure that checks for fits do not affect later checks, we must encapsulate these side-effects.

Using Quantification: My first (naive) approach to avoid side-effects was to wrap the type with any givens from the implications and quantifying any free type variables, which meant that any effects on the variables only affected fresh variables introduced by the type checker during simplification. However, this approach rejected some valid hole fits and

accepted some invalid hole fits since the type `forall a. a` is not equivalent to `a` in most cases.

Using a Wrapper: The current approach to avoid side-effects uses a wrapper that restores flexible meta type variables back to being flexible after the operation has been run, reverting any side-effects on those variables.

4 An Additional Application

The reason I started looking into valid hole fits for typed-holes was to be able to interact with libraries of functions annotated with non-functional properties.

A Library of Sorting Algorithms annotated with computational complexity and memory complexity is one example. We can define a type to represent simple asymptotic polynomials for a simplistic encoding of big *O* notation:

```
{-# LANGUAGE TypeInType, TypeOperators, TypeFamilies,
             UndecidableInstances, ConstraintKinds #-}
module ONotation where

import GHC.TypeLits as L
import Data.Type.Bool
import Data.Type.Equality

-- Simplistic asymptotic polynomials
data AsymP = NLogN Nat Nat

-- Synonyms for common terms
type N     = NLogN 1 0
type LogN  = NLogN 0 1
type One   = NLogN 0 0

-- Just to be able to write it nicely
type O (a :: AsymP) = a

type family (^.) (n :: AsymP) (m :: Nat) :: AsymP where
  (NLogN a b) ^. n = NLogN (a L.* n) (b L.* n)

type family (*.) (n :: AsymP) (m :: AsymP) :: AsymP where
  (NLogN a b) *. (NLogN c d) = NLogN (a+c) (b+d)

type family OCmp (n :: AsymP) (m :: AsymP) :: Ordering where
  OCmp (NLogN a b) (NLogN c d) =
    If (CmpNat a c == EQ) (CmpNat b d) (CmpNat a c)

type family OGEq (n :: AsymP) (m :: AsymP) :: Bool where
  OGEq n m = Not (OCmp n m == 'LT)

type (>=.) n m = OGEq n m ~ True
```

We can now annotate a library of sorting functions to use *O* notation to convey complexity information:

```
{-# LANGUAGE TypeInType, TypeOperators, TypeFamilies,
             TypeApplications #-}
module Sorting ( mergeSort, quickSort, insertionSort
               , Sorted, runSort, module ONotation) where

import ONotation
import Data.List (insert, sort, partition, foldl')

-- Sorted encodes average computational and auxiliary
-- memory complexity. The complexities presented
-- here are the in-place complexities, and do not match
-- the naive but concise implementations included here.
newtype Sorted (cpu :: AsymP) (mem :: AsymP) a
              = Sorted {runSort :: [a]}

insertionSort :: (n >=. O(N^.2), m >=. O(One), Ord a)
              => [a] -> Sorted n m a
insertionSort = Sorted . foldl' (flip insert) []

mergeSort :: (n >=. O(N*.LogN), m >=. O(N), Ord a)
          => [a] -> Sorted n m a
mergeSort = Sorted . sort

quickSort :: (n >=. O(N*.LogN) , m >=. O(LogN), Ord a)
          => [a] -> Sorted n m a
quickSort (x:xs) = Sorted $ (recr lt) ++ (x:(recr gt))
  where (lt, gt) = partition (< x) xs
        recr = runSort . quickSort @(O(N*.LogN)) @(O(LogN))
quickSort [] = Sorted []
```

Using valid hole fits, we can then search the sorting library by specifying the desired complexity in the type of a hole to find functions with those properties (or better):

```
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
```

Figure 4. Valid hole fits found in GHCi version 8.6 for the hole in `_ [3,1,2] :: Sorted (O(N*.LogN)) (O(N)) Integer`

5 Related Work & Ideas

Hoogle is *the* type directed search engine for Haskell, and allows users to easily search all of Hackage for functions by type or name [12]. Hoogle, however, does not integrate with the type checker of GHC, and can have difficulties with handling complex types and type families. Hoogle uses data extracted from the Haddock generated documentation of packages [12], meaning that unexported functions in the current, local module and local bindings like function arguments and bindings defined in **`let`** or **`where`** clauses are not discoverable. For searching the Haskell ecosystem however, Hoogle remains unparalleled.

Program Synthesis: Finding valid hole fits can be considered a special case of type-directed program synthesis. **Djinn** is a program synthesis tool that generates Haskell code from a type, and can generate total functions rather than just single identifiers fom user provided types and functions [2]. **Synquid** is a command line tool and algorithm that can synthesize programs from polymorphic refinement types in an ML-like language [14]. Other program synthesis tools include **InSynth** and **Prospector** [6, 11], however none of these are integrated with a compiler or type checker of a language, but are rather stand-alone tools or IDE plugins.

PureScript: The valid hole fits as presented in this report are modeled on the type directed search that Hegemann implemented in PureScript as part of his Bachelor's thesis

work [8]. In PureScript, the type directed search looks for matches when a typed-hole is encountered [8]. The valid hole fits as I have implemented them in GHC go further than those in PureScript in that the output is sorted, and additional arguments are available via refinement hole fits.

Agda: The typed-holes of GHC were originally inspired by Agda [7]. Agda is dependently typed, and thus can offer very specific matches. The emacs mode of Agda offers the `Auto` command to automatically fill a hole with a term of the correct type, and the `Refine` command can split a hole into cases containing additional holes [1]. The dependent typing has the drawback that type inference is in general undecidable, and users must explicitly provide more types than required in Haskell [13].

Idris, like Agda, is dependently typed, and offers a proof-search command that can construct terms of a given type [3]. Idris also has a type directed search command, but in Idris the command also gives (and denotes) matches with a more specific type, in addition to matches of the same or more general type [3]. This allows users to find functions that match `Eq a => [a] -> a` when searching for `[a] -> a`, even though it requires an additional constraint [3]. Idris does not integrate these commands with typed-holes.

6 Conclusion

As can be seen from the examples in this report, valid hole fits can be useful in many different scenarios. They can improve the user experience for Haskell programmers working with prelude functions like `foldl` or advanced features like `lens` or `TypeInType`. The implementation makes use of the already present type-checking mechanisms of GHC, and integrates well with typed-holes in a non-intrusive manner. I believe it to be good addition to the typed-holes of GHC; it should help facilitate Type-Driven Development in Haskell.

I learned a great deal from this project. Extending GHC was certainly non-trivial, however, the modularity of GHC allowed me to reuse a lot of code and to focus on the *what* rather than the *how*. A few pitfalls were encountered (like type checking by side-effect), and while the documentation of GHC internals is not so great (being mostly spread around in comments and assuming a lot of knowledge from the reader), the community was very helpful to a newcomer.

6.1 Future Work

When working with typed-holes, a few issues come to light:
Too General Fits: The types inferred by GHC are sometimes too polymorphic for the valid hole fits to be useful. One such example is if we consider the function `f x = (_+x)/5`. Here, GHC will happily infer the most general type, namely that `f :: Fractional a => a -> a`. A sensible hole fit for the hole in `f` is `pi :: Floating a => a`, but that would constrain `f` to the more specific type of `Floating a => a -> a`. If `f` is not explicitly typed, then `pi` should be a valid hole fit. However, `f` having a more specific type might invalidate other code that uses `f`, if those uses are explicitly typed with a `Fractional` constraint and not a `Floating` constraint. We would like to suggest such hole fits, for example by including a list of more specific hole fits, such as offered by Idris [3].

Built-in Syntax: Functions that are built-in syntax are not considered as candidate hole fits, since they are not in the global reader. However, functions like `(,)`, `[_]`, and `(:) :: a -> [a] -> [a]` are very common, and suggesting them would improve the user experience. Since these functions are syntax, they are not "in scope" in the global reader and no list of these functions is defined in GHC, making the addition of built-in syntax candidates non-trivial. One solution would be to hard-code these as candidates.

Functions with Fewer Arguments: There is no way to find functions that take in fewer arguments than required, and users must resort to binding the arguments (with e.g. `(\x -> _)`) in order to find these suggestions. Considering lambda abstractions as candidates could improve this case.

Specifying Behavior: It can be hard to choose which fit to use when multiple fits with the right type but different behaviors are suggested. Being able to hint to GHC how the function should behave would allow us to discard wrong hole fits. One approach would be integrating the valid hole fits with something like the refinement types of Liquid Haskell:

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

in which users can specify invariants for behavior [15].

6.2 Current Status

My contributions to GHC have been accepted. A basic version of the valid hole fits is in GHC version 8.4, an improved version with sorting, refinement hole fits and local binding suggestions in GHC version 8.6, and on GHC HEAD, a version is available with a flag to display documentation for hole fits in the output (to explain opaque function names). All code is available in the `TcHoleErrors` module in GHC.

Acknowledgments

I would like to thank the members of the Haskell community who helped me while writing my extensions to GHC. In particular I would like to thank Simon Peyton Jones for sharing his intimate knowledge of the intricacies of GHC with me, Oleg Grenrus for suggesting sorting by subsumption, Ben Gamari and Matthew Pickering for reviewing my contributions and for their feedback and suggestions, Mary Sheeran for her supervision, and Koen Claessen and Sólrún Halla Einarsdóttir for their feedback on this report.

A SEED project from the Chalmers Area of Advance ICT provided the initial impetus for this work. This work was partially funded by the Swedish Foundation for Strategic Research (SSF) under the project Octopi (Ref. RIT17-0023).

References

[1] Agda Contributors. 2017. Agda Documentation 2.5.3. https://agda.readthedocs.io/en/v2.5.3/

[2] Lennart Augustsson. 2014. The Djinn package. https://hackage.haskell.org/package/djinn

[3] Edwin Brady. 2017. *Type-Driven Development with Idris*. Manning Publications Company.

[4] GHC Contributors. 2017. GHC 8.2.1 users guide. https://downloads.haskell.org/~ghc/8.2.1/docs/html/users_guide/index.html

[5] GitHub. 2017. The Open Source Survey. http://opensourcesurvey.org/2017/

[6] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *PLDI '13, Proc. the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 27–38.

[7] Haskell Wiki Contributors. 2014. Typed holes in GHC. https://wiki.haskell.org/index.php?title=GHC/Typed_holes&oldid=58717

[8] Christoph Hegemann. 2016. Implementing type directed search for PureScript. (2016). BSc. Thesis, University of Applied Sciences, Cologne.

[9] Graham Hutton. 2016. *Programming in Haskell*. Cambridge University Press.

[10] Edward Kmett. 2018. The lens library. https://hackage.haskell.org/package/lens

[11] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *PLDI '05, Proc. the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 48–61.

[12] Neil Mitchell. 2008. Hoogle overview. *The Monad. Reader* 12 (2008), 27–35.

[13] Ulf Norell. 2008. Dependently typed programming in Agda. In *International School on Advanced Functional Programming*. Springer, 230–266.

[14] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *PLDI '16, Proc. the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 522–538.

[15] Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *ICFP '14, Proc. the 19th ACM SIGPLAN International Conference on Functional Programming*. ACM, 269–282.