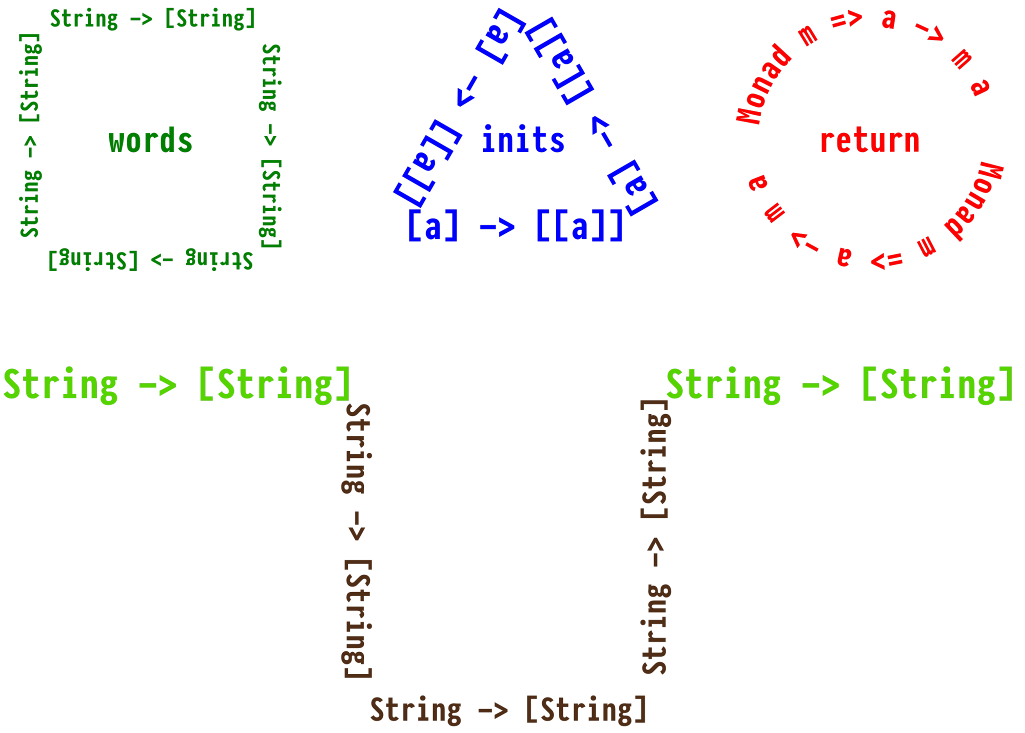
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Suggesting Valid Hole Fits for Typed-Holes in Haskell

Master's thesis in Computer Science - Algorithms, Languages, and Logic

MATTHÍAS PÁLL GISSURARSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Suggesting Valid Hole Fits for Typed-Holes in Haskell

MATTHÍAS PÁLL GISSURARSON

Department of Computer Science and Engineering
Functional Programming Division
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Suggesting Valid Hole Fits for Typed-Holes in Haskell
MATTHÍAS PÁLL GISSURARSON

Supervisor: Mary Sheeran, Department of Computer Science and Engineering
Examiner: Alejandro Russo, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Functional Programming Division
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An illustration of a typed-hole and three candidate functions which all fit the hole.

Typeset in LaTeX
Gothenburg, Sweden 2018

Suggesting Valid Hole Fits for Typed-Holes in Haskell
MATTHÍAS PÁLL GISSURARSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Most programs are developed from some sort of specification, and type systems allow programmers to communicate part of this specification to the compiler via the types. The types can then be used to verify that the implementation matches this partial specification. But can the types be used to aid programmers during development, beyond verification? In this thesis I present a lightweight and practical extension to the typed-holes of GHC that improves user experience and facilitates a style of programming called "Type-Driven Development".

Útdráttur

Flest forrit eru þróuð útfrá einhverskonar hönnunarlýsingu á þeim kröfum sem gerðar eru til forritsins, en tögunarkerfi leyfa forriturum að miðla hluta af þessari lýsingu til þýðandans í gegnum tög. Þýðandinn getur svo notað þessi tög til þess að staðfesta að útfærslan sé í samræmi við hlutlýsinguna sem tögin tjá. En er hægt að nota þessi tög til þess að styðja við bakið á forriturum á meðan á þróun stendur, fremur en bara til staðfestingar í lokin? Í þessari ritgerð kynni ég til sögunnar létta og hagnýta útvíkkun á töguðum holum sem samþykkt hefur verið í GHC sem bætir upplifun notenda og greiðir fyrir forritunarhætti sem kallast "Tagaknúin þróun".

Keywords: Typed-Holes, Haskell, GHC, Type-Driven Development, Functional Programming

Acknowledgements

First of all, I would like to thank my supervisor Mary Sheeran for her patience during my long journey, especially her numerous reads of drafts and helpful comments there upon. I would also like to thank my examiner, Alejandro Russo, for helpful comments on my final draft and pointing me towards work on program synthesis, which I had been unaware of and neglected to include as related work.
I would also like to express my gratitude to 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 taking the time to meet with me at ICFP and later at Chalmers and sharing his intimate knowledge of the intricacies of the GHC type checker, and Ben Gamari and Matthew Pickering for reviewing my contributions to GHC and providing feedback and suggestions.
I would like to express my thanks to Christoph Hegemann, who implemented the type-directed search in PureScript, which inspired my main contribution in this thesis.
Finally, I'm grateful for my friends and family for their company during my masters studies. I'm especially grateful to my fantastic and hard-working girlfriend Sólrún Halla Einarsdóttir for taking the time out of her busy schedule to proofread and comment on my drafts and providing moral support during stressful periods of binge writing. Takk ástin mín!

Matthías Páll Gissurarson, Gothenburg, June 2018

Contents

1 Introduction

Most programs are developed from some sort of *specification*: there is an idea about what the program is supposed to do, but whether that idea is clearly stated in a technical document or just a vague idea in the mind of the developer depends on the environment and the rigour demanded of the software itself. This raises an important question: does the implementation of the program fulfill the specification? This can be checked by testing, but a more rigorous way to *verify* that the implementation of the program fulfills the specification is to use *type systems*.

Type systems allow the programmer to tightly couple parts of the specification of a program and its implementation by annotating their program with type signatures. This partial specification can help the programmer avoid a wide variety of bugs and allows the compiler to verify that the program behaves according to the part of the specification encoded in the types [1]. It is important to note that usually only part of the specification can be expressed in the type system, so while types play an important role, they can only guarantee correctness up to the level of expressiveness that the type system in question allows [1].

The research question I seek to answer in this thesis is the following: can we take further advantage of the tight coupling between this partial specification described by the types and the implementation, and use it to improve the user experience during development? User experience is the third most important factor when choosing open source software, after stability and security, so an improved user experience would be a valued improvement overall [2]. In this thesis I present a lightweight and practical extension to the typed-holes of GHC, which leverages the already present type checking mechanisms to improve the user experience when developing Haskell programs and facilitates a style of programming called "type-driven development (TDD)". It is lightweight compared to an interactive theorem prover and practical in the sense that it is both easy to use and performs well, even for non-total programs (like most Haskell programs).

In type-driven development, the programmer first writes the types of the various parts of the program, and then writes the implementations of those parts. This allows the programmer to "fill in the puzzle", and to use the types to guide their implementation [3].

The "Glasgow Haskell Compiler (GHC)" already facilitates this style of development with the typed-holes feature. By using a typed-hole, the programmer can ask GHC what type a particular expression is inferred to have, which helps them in figuring out the "shape" of the next "piece of the puzzle" [4].

However, this still relies on the programmer to figure out what pieces are available to them. When dealing with complex types or unfamiliar libraries, this can prove to

be a challenge, especially to beginners. By extending the typed-holes functionality of GHC, we can improve the situation and facilitate TDD even in the presence of these complex types or unfamiliar libraries. We do this by presenting a list of "valid hole fits" when a typed-hole is encountered.
Valid hole fits are expressions that are valid fits in place of the hole, which is to say that replacing the hole with the fit will result in a type correct program. By using the information that the programmer has already imparted or the compiler inferred, these fits can be used to guide the implementation of a specified program. Additionally, this extension can enable libraries of functions with the same functional type, but different non-functional properties, to be used in a practical way for development, as explored in the evaluation.

1.1 Why Haskell and GHC?

The valid hole fits are implemented as an extension to GHC for Haskell.

1.1.1 Pros

Haskell has a wide adoption in both academia and industry [5]. It is one of the more widely used strongly-typed functional programming languages, according to the 2017 Stack Overflow developer survey [5]. The only other strongly-typed functional programming language in that survey was Scala, [5]. However, Scala does not have typed-holes, and the type-inference used by Scala requires the programmer to provide more type annotations than the extended Hindley–Milner type system that GHC implements for Haskell [6, 7].
GHC was chosen since it is the most widely used compiler for Haskell by far [8], and already provides typed-holes which we can piggyback on to provide valid hole fits for those holes.
Improving the type-driven development approach in GHC for Haskell would help users gain additional value and improve their experience when using the strong types of Haskell, without losing access to the wealth of libraries and study materials available for Haskell.

1.1.2 Cons

As we will see in the evaluation, Haskell's type system is not as expressive as we would like. Having more expressive types would improve the valid hole fits extension quite a bit. Then the user can impart more knowledge about the desired outcome to the compiler. One way of having more informative types is to have dependent types, where the type of a value is allowed to depend on the value itself, as implemented in Agda and Idris [9, 10, 3]. Another way is to use refinement types, where invariants can be encoded as a combination of types and predicates, such as implemented in a restricted way in Liquid Haskell [11].
Haskell does not have fully dependent types at the time of writing, but extensions such as type families and Generalized Algebraic Datatypes (GADTs) allow for type-level computation, which allows for some approximation of fully dependent types

[11]. An extension to Haskell that allows for fully dependent types is in development, and is expected to be ready in 2021 [12].

1.2 Preliminary Background

This chapter provides preliminary background material on type systems, Haskell, GHC and typed-holes, with more detailed background on the relevant parts of GHC and the underlying type theory in Chapter 2.

1.2.1 Type Systems

Type systems were originally formalized by Russell and Whitehead as a way to avoid paradoxes that threatened the foundations of mathematics. They have since become standard tools in logic and computer science for proofs and the verification of programs [1]. Type systems can be used to couple the specification of the program to the implementation, by annotating functions with *types*. This allows the programmer to communicate their intent to the compiler, and allows the compiler to check whether the implementation does indeed fulfill the partial specification specified by the types.
One specific type system, the Hindley–Milner type system, is of particular interest. It is one of the most popular and widely studied type systems, and allows for global inference of the types of untyped programs [13]. This means that programmers of languages using Hindley–Milner as the basis for their type system (such as Haskell and ML) need not provide a type to every expression. This makes using strong static type system such as the Hindley–Milner system practical [13].

1.2.2 Type-Driven Development (TDD)

Similar to test-driven development, type-driven development is an approach to the development of programs in which the programmer writes the types of their programs first, before they write the implementation of the program. This allows the types to guide the development, rather than tacking them on afterwards only to verify the correctness of the implementation [3].

1.2.3 Haskell & GHC

Haskell is a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static typing [14]. Haskell's type system is a an extension of the Hindley–Milner type system with type classes, type equality coercions and constraints called System F_C [15, 16].
The *Glasgow Haskell Compiler* is a state-of-the-art, open-source implementation of Haskell, with two main components: the batch compiler *GHC*, and GHCi, an interactive interpreter [17]. It is the most widely used compiler for Haskell [8]. The Glasgow Haskell compiler implements a number of extensions, many of which have later been incorporated into the updated standard for Haskell (Haskell 2010). One of these extensions is the typed-holes functionality [18].

1.2.4 Typed-Holes

Typed-holes in *GHC* were introduced in version *7.8.1* and implemented by Simon Peyton Jones, Sean Leather and Thijs Alkemade [4]. 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 any relevant local bindings [18].

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

Figure 1.1: An example of a hole

As an example, the function `f` in figure 1.1 has a hole in place of a function application. When this code is compiled, GHC will output a *hole error*:

```
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)
```

This message tells us quite a lot about what GHC knows or can infer about the hole, like where it is, what its inferred type is and what local bindings are in scope that mention any of the types mentioned in the hole.

1.3 Valid Hole Fits

My contribution in this thesis is to extend the typed-hole error message with a list of valid hole fits. When the code in figure 1.1 is compiled with the latest version of GHC, it will in addition to the previous message output a list of "valid hole fits" like the following:

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

Figure 1.2: The valid hole fits for the hole in figure 1.1

These valid hole fits tell the user what bindings are in scope that they could use in place of the hole, their type, and what types the quantified type variables in their type would be if used in place of the hole.

1.3.1 Valid Refinement Hole Fits

While showing only valid hole fits that fit the hole exactly can be useful, especially for beginners, it is often the case that the hole would need to be filled with a function applied to some additional arguments. For example if we compile the following:

```haskell
f :: Num a => [a] -> a
f = _
```

Figure 1.3: A hole with a type class constraint where `fold` would be a good match.

The valid hole fits will be:

```
Valid hole fits include
  f :: [a] -> a
  head :: forall a. [a] -> a
    with head @a
  last :: forall a. [a] -> a
    with last @a
  product :: forall (t :: * -> *). Foldable t =>
             forall a. Num a => t a -> a
    with product @[] @a
  sum :: forall (t :: * -> *). Foldable t =>
         forall a. Num a => t a -> a
    with sum @[] @a
```

Additionally, the type of `f` includes as a given that the type `a` satisfies the constraint `Num a`. When a quantified type variable such as `a` must satisfy some constraints, the typed-hole error message will include a message that relates those constraints. For the hole in `f`, it will state:

```
Constraints include Num a (from hf.hs:3:1-22)
```

If GHC is passed the flag `-frefinment-level-hole-fits=N`, the message is further extended with a list of valid refinement hole fits. These are valid hole fits with additional holes in them. If the code in figure 1.3 is compiled with the flag where $N = 2$, the valid hole fits will be the following:

```
Valid refinement hole fits include
  foldl1 (_ :: a -> a -> a)
    with foldl1 @[] @a
    where foldl1 :: forall (t :: * -> *). Foldable t =>
                    forall a. (a -> a -> a) -> t a -> a
  foldr1 (_ :: a -> a -> a)
    with foldr1 @[] @a
    where foldr1 :: forall (t :: * -> *). Foldable t =>
                    forall a. (a -> a -> a) -> t a -> a
  foldl (_ :: a -> a -> a) (_ :: a)
    with foldl @[] @a @a
    where foldl :: forall (t :: * -> *). Foldable t =>
                   forall b a. (b -> a -> b) -> b -> t a -> b
  foldr (_ :: a -> a -> a) (_ :: a)
    with foldr @[] @a @a
    where foldr :: forall (t :: * -> *). Foldable t =>
                   forall a b. (a -> b -> b) -> b -> t a -> b
  const (_ :: a)
    with const @a @[a]
    where const :: forall a b. a -> b -> a
  ($) (_ :: [a] -> a)
    with ($) @'GHC.Types.LiftedRep @[a] @a
    where ($) :: forall a b. (a -> b) -> a -> b
  (Some refinement hole fits suppressed;
   use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)
```

where N controls how many additional holes we look for. These refinement hole fits can help guide the programmer to a type correct implementation of their specification, as specified by the types.

1.4 Motivating Examples

The following are a few scenarios for which the user experience can be improved. We use Hoogle, the Haskell API search as a baseline. For more information on Hoogle, see section 5.2.

1.4.1 Working with the Prelude

Recall that when we compile the following code with GHC 8.2.1:

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

The error message generated by the hole in `f` will be:

```
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)
```

When we look up the type of the hole as inferred by GHC, `[Char] -> [String]` in Hoogle, the first 6 results are:

- `pyIncludes :: String -> [[Char]]` from `matplotlib`,
- `unintersperse :: Char -> String -> [String]` from `Cabal`,
- `splitSepList :: Char -> String -> [String]` from `bibtex`,
- `split :: Char -> String -> [String]` from `ghc`,
- `sepList :: Char -> [String] -> String` from `bibtex`, and
- `namedNumbers :: [Char] -> String -> [[Char]]` from `xmonad-contrib`.

Only one of these results has the exact same type, and none is likely to be the one that the user is looking for. If we change the type to `String -> [String]`, the expected `lines` and `words` from `base` show up as the first two suggestions, but a beginner might not realize that they need to change the type in order to get more relevant results. To improve the user experience, we would like to show relevant functions that are in scope, which fit the hole and work even with type synonyms at play.

1.4.2 The Lens Library

For a more advanced scenario, consider the following code which uses the `lens` library. The `lens` library is a library that allows users to define “lenses”, which are functions that allow users to “zoom-in” on parts of a data structure and operate on that part without having to consider the rest of the data structure [19].

```
module LensDemo where

import Control.Lens
import Control.Monad.State

newtype Test = Test { _v :: Int }
    deriving (Show)

val :: Lens' Test Int
val f (Test i) = Test <$> f i

updTest :: Test -> Test
updTest t = t &~ do
    _ val (1 :: Int)
```

In the code above, we define a simple data structure, `Test`, which contains a single field of type `Int`, and we define a simple lens `val` to operate on that field. We then want to write a function that updates `val` with the value 1 using do-notation, but we are unsure how. To get a sense of what we can do, we add a hole, and compile. When we compile this code with GHC 8.2.1, we get the following output:

```
Found hole:
  _ :: ((Int -> f0 Int) -> Test -> f0 Test) -> Int -> State Test a0
Where: ‘f0’ is an ambiguous type variable
       ‘a0’ is an ambiguous type variable
In the expression: _
In a stmt of a ’do’ block: _ value (1 :: Int)
In the second argument of ‘(&~)’, namely ‘do _ value (1 :: Int)’
Relevant bindings include
  t :: Test (bound at src/test.hs:12:9)
  updTest :: Test -> Test (bound at src/test.hs:12:1)
```

As we can see, the type of the hole is quite complex, involving multiple ambiguous type variables and a monad transformer. Copying this to Hoogle reveals no useful suggestions, with the top 6 being:

- `mapState :: ((a, s) -> (b, s)) -> State s a -> State s b`
  from `transformers`
- `mapState :: () => ((a, s) -> (b, s)) -> State s a -> State s b`
  from `intro`,
- `mapState :: ((a, s) -> (b, s)) -> State s a -> State s b`
  from `rev-state`,
- `lift :: Monad m => T r s -> State s a -> StateT r m a`
  from `data-accessor`,

-
  ```
  scanner :: Parser token result -> Scanner st token
          -> Scanner (State st token result) result
  ```
  from `yi-core`, and

-
  ```
  step :: Methods queue set -> State queue label set
       -> [State queue label set]
  ```
  from `set-cover`.

None of these results is from the `lens` package, and none is relevant to the current context. To improve the user experience in these cases, we would like to get more useful suggestions that pertain to the types involved.

2
Background

This chapter contains an overview of the parts of the GHC type checker that are relevant to our implementation of typed-holes and their underlying theory. All the concepts related here play a part in the implementation of valid hole fits in GHC, as detailed in Chapter 3 and are presented as clarification of how they operate in the context of GHC. Much of the documentation of the type checker and its parts is spread throughout the GHC source code in various comments, so this chapter also serves as an external academic reference of these concepts as they are used in the context of GHC. For reference, we will use the source code for GHC version 8.4.2.

2.1 The Type Checker

Haskell is a language with a strong static type system that is an extension of the Hindley–Milner type system [14, 15, 16]. The type system of Haskell defines a set of base types, such as `Int` and `Char`, along with methods for combining base types into more complex types, like function types. It also allows the construction of new types using type constructors like `Maybe` and `IO`, and even allows the user to define their own type constructors [14]. In fact, the base types are implemented as unary type constructors (known as "type constants") [14]. Users can specify the types of variables and expressions by giving them a *type signature* of the form `v :: T`, where `v` is a variable (usually a top-level function) or expression and `T` is a type in the type system [14].

When GHC compiles a Haskell program, it does type checking (including type inference) in order to verify that the program is type-correct [20]. For a program to be type-correct, it must pass the type checker [20]. A type checker has two functions:

- Type checking, which takes an expression e and a type T and decides whether e has type T, denoted with $e : T$ [21].
- Type inference, which takes an expression e and finds a type T such that $e : T$ [21].

A program passes the type checker if it is "well-typed", i.e. if the type checker can infer a type for each expression and variable in the program [1]. Since GHC allows users to specify the types of variables and expressions, the type that the type checker infers (the "actual" type) must match the type given by the user (the "expected" type) [22]. The GHC type checker does type inference in two steps [23, 20]. First it generates a collection of type constraints from the source syntax tree, and then solves the gathered constraints [23, 20].

2.2 Type Variables

Type variables are uninterpreted base types which can be substituted or instantiated with other types [1]. A type substitution is a mapping from type variables to types, and applying a substitution σ to a type variable T to obtain a type σT is called instantiating the type variable [1]. They are used to allow for polymorphism on the type level [1]. In the type checker, there are two main variants of type variables, *Skolems* and *Meta type variables*. A third variant is a *Runtime unknown*, which is used in the GHCi interactive context [22].

2.2.1 Skolems

Named after the similar process of converting a first order logic formula to skolem normal form in mathematical logic, *Skolemization* in the context of GHC is when a type is decomposed and fresh type constants are introduced for all quantified type variables in the type and the type with all its arrows visible (i.e. not buried under `foralls`) is returned. [22, 24]. The fresh type constants introduced during skolemization are called *skolem* type variables (or just *skolems*) [7].
As an example, when the type `Int -> forall a. Ord a => blah` is skolemised, the result of skolemization is the type `Int -> blah`, a list of skolems `[a]`, the givens for those skolems `[d: Ord a]` and a `HsWrapper` (see section 2.7.1 for more information about wrappers) which GHC can use to go from an expression of the return type to an expression of the original type [22].

2.2.2 Meta Type Variables

A *Meta type variable* is a unification type variable, used by the type checker to track unifications. It has a level number to track when it is touchable by the type checker and an `IORef` that the type checker uses to unify the variable with some type [22]. See section 2.3 and section 2.4 for more details.

2.3 Constraints

Type inference such as in Haskell can be carried out in two stages, by first generating constraints from the source program, and then solving those constraints without regard to the source program [23]. This is done by using the previously mentioned unification variables, which stand for uninterpreted types, and solving the constraints produces a substitution that assigns a type to each unification variable [23]. The most basic type of constraint is the type equality constraint of the form $t_1 \sim t_2$ where t_1 and t_2 are types [23]. Another common type of constraint is the type class constraint of the form $C\ t$, where C is a type class and t is a type [22]. To do type-inference for GADTs, we need a new sort of constraint, namely implication constraints [23].
In the following, we will use as an example the constraints that arise when looking for valid hole fits for the hole in `f` in the following piece of code:

```
f :: forall a b. (a ~ b, Show b) => a -> String
f x = show _
```

Figure 2.1: Example of a hole with both given and imposed constraints.

2.3.1 Implication Constraints

An implication constraint is a set of wanted constraints with additional information and invariants such as skolems, given constraints, and the level number of the implication constraint [22]. An implication constraint also has a `TcEvBinds` field which contains the `EvBindsVar` which tracks the `EvBindsMap` of evidence during constraint generation and simplification [23, 22]. See section 2.5 for more information on the `EvBindsMap`.

2.3.1.1 Skolems

The set of skolems of an implication constraint is the set of type constants introduced during the skolemization of the types involved in the constraints [22]. In our example in figure 2.1, there are two skolems, `a_a1q2` and `b_a1q3`, which are the constant used to represent the `forall` quantified `a` and `b` in the type of `f`.

2.3.1.2 Given Constraints

The set of givens of an implication is the set of constraints which are known to hold in the local context of the implication [23]. These can be constraints such as type equality constraints and class constraints, and other kinds of constraints. They are represented by evidence variables in GHC [23, 22].
In our example in figure 2.1, the givens are `Show b_a1q3` and `a_a1q2 ~ b_a1q3`.

2.3.1.3 Wanted Constraints

The wanted constraints is the set of constraints that we want to solve. They need to be deducible from the given constraints for the implication constraint to be soluble [23]. They are split into two sets, a set of "simple" constraints and a set of "proper" constraint [23]:

- The set of simple constraints is the set of implication constraints that do not involve any local equalities, i.e. the set of constraints which do not have any additional givens that are valid only locally in that constraint [23].
- The proper constraints are a set of constraints that contain additional local equalities, which means that they are implication constraints with additional givens (and possibly skolems) [23].

In our example, the wanted constraints contain only simple constraints: A type class constraint `Show a_a1q7`, and a hole constraint, `a_a1q7`. A hole constraint is a constraint used to represent a typed-hole, and consists of a unification variable which must be unified in order to solve the hole [22].

2.4 Type Checker Levels

Each meta type variable and implication has a level number, which is used to track invariants and whether a meta type variable is touchable [22]. These invariants are:

- Implication Invariant: The level number of an implication is strictly greater than (>) that of its parent [22].
- Given Invariant: The level number of a unification variable appearing in the givens of an implication is strictly less than (<) the level number of the implication it appears in [22].
- Wanted Invariant: The level number of a unification variable appearing as a wanted in an implication is less than or equal to (≤) the level number of the implication it appears in [22].

A unification variable is touchable if its level number is equal to (=) that of its immediate parent implication [22]. During unification, we only unify touchable unification variables [22].

2.5 Evidence

An evidence term is a term that witnesses that a constraint holds, and is, conceptually, an expression in GHC Core that implements the constraint [23, 22].
Evidence terms take a few different forms depending on what kind of constraint they witness, with the main two forms being type equality coercions for witnessing type equalities and dictionaries for witnessing type class constraints [23, 22]. Additional forms are special cases including dictionaries for witnessing `Typeable` and `HasField` constraints, and type coercions for witnessing explicit casts [22].

2.5.1 Evidence Variables

Evidence variables are used to track evidence during type checking, where the `TcEvBinds` of an implication contains a mutable reference to an `EvBindMap`, which maps evidence variables to evidence binds [22]. Evidence binds contain an evidence term and whether this evidence is given or wanted [22]. During simplification, the solver updates the `EvBindsMap` and maps the evidence variables in the wanted constraints to evidence binds, which are later used to provide the evidence needed for the Core expressions [25, 22].

2.6 Simplification

GHC uses a simplifier to solve a set of wanted constraints. It first solves all the simple constraints, matching evidence variables to givens and creating a substitution for those evidence variables by filling up the `EvBindMap`. It then solves the proper implications by using the substitution found when solving the simple constraints [25, 22]. Simplification is run at a given type checker level which is controlled by the `TcM`, the type checker monad, which controls which unification variables are touchable [22].

2.7 System F_C (GHC Core)

GHC uses an intermediate language called System F_C during compilation, which is an extension of the polymorphic lambda calculus known as System F with support for non-syntactic type equality [22, 15]. Often referred to as Core within GHC, it allows GHC to operate on a much simpler but equivalent language which is easier to type check and optimize than the source Haskell, with most optimizations being expressed as Core-to-Core transformations [22]. Core uses explicit evidence-passing for passing evidence such as type equalities or type class constraints [23, 22].

2.7.1 HsWrapper

A *HsWrapper* is a wrapper that contains information on how to go from one expression to another in GHC Core [22]. As an example, if `e` is an expression and we have a wrapper

```
wp = \x :Int -> /\ a. \(d: Ord a) . <hole> x,
```

then

```
wp e = \x : Int -> /\ a. \(d: Ord a). e x,
```

is obtained by replacing the `<hole>` with the expression `e`. Note that type class arguments are explicit in Core, as shown with the `\(d: Ord a)` in the wrapper above [22]. If you have an `e :: Int -> blah`, then the application of `wp` to `e` is `wp e :: Int -> forall a. Ord a => blah`, and `wp` will bind any type variables and givens [22]. Here, `<hole>` is the identity wrapper, and not related to typed-holes.

2.8 Zonking

GHC relies heavily on mutability in the type checker for efficient operation. For this reason, throughout much of the type checking process meta type variables are represented by mutable variables (known as TcRefs) [22].
Zonking is the process of traversing the entire type expression and of ripping out these mutable variables and replacing them with a real type. An unfilled type variable is either ignored (such as in the middle of type checking) or replaced with a value like `Any` (which means that it could be any type) [22]. These "real" types might be type variables, so when we zonk before outputting we do so in a `TidyEnv`, which ensures that if a type contains any free type-variables, these are prettified and added to the environment. If we later use this environment to zonk another type before outputting that contains a free type variable that we have already encountered, the tidy environment ensures that it is zonked to the same representation [22].

3 Implementation

In this chapter, I will go into detail on how we prepare a set of constraints and invoke the type checker with the types of various global reader elements and local identifiers to find valid hole fits for a given hole. For explanation of the various terms and parts, refer to the background material on the relevant parts of the GHC type checker in chapter 2. The code for the implementation is given in the appendix in section A.1.

3.1 Finding Valid Hole Fits

The valid hole fit suggestions for typed-holes in GHC are implemented as an extension to the error reporting mechanism of GHC.

When GHC encounters a hole, it creates a hole constraint within the type checker. Holes are treated as type errors, and all reporting of type errors is postponed until the end of the compilation. This allows the compiler to continue type checking the rest of the program, to gather as much information as it can and report as many errors as it can.

We look for valid hole fits when the error reporting mechanism for a typed-hole is invoked. This means that we are free to put an emphasis on utility over performance, since the performance cost is only incurred when the code would not have compiled in any case. However, we do take care to not be too slow, since we want typed-holes and valid hole fits to be viable in an interactive environment like GHCi.

The process is roughly as follows (as visualized in 3.1):

1. Gather candidates from the context (local and global).
2. Gather constraints from the context that are relevant to the hole.
3. Filter the candidates by the following:
   (a) Generate a subtype wrapper that maps the type of the candidate to the type of the hole, and gather any emitted constraints.
   (b) Invoke the type checker/constraint solver to check whether the associated constraints along with the relevant constraints are soluble, given any givens associated with the hole.
   (c) Accept those candidates where the constraints are soluble, since that means that the candidate fits the hole.
4. Sort the fits found according to the size of type application or by subsumption.
5. Append a list of valid hole fits to the typed-hole error message.

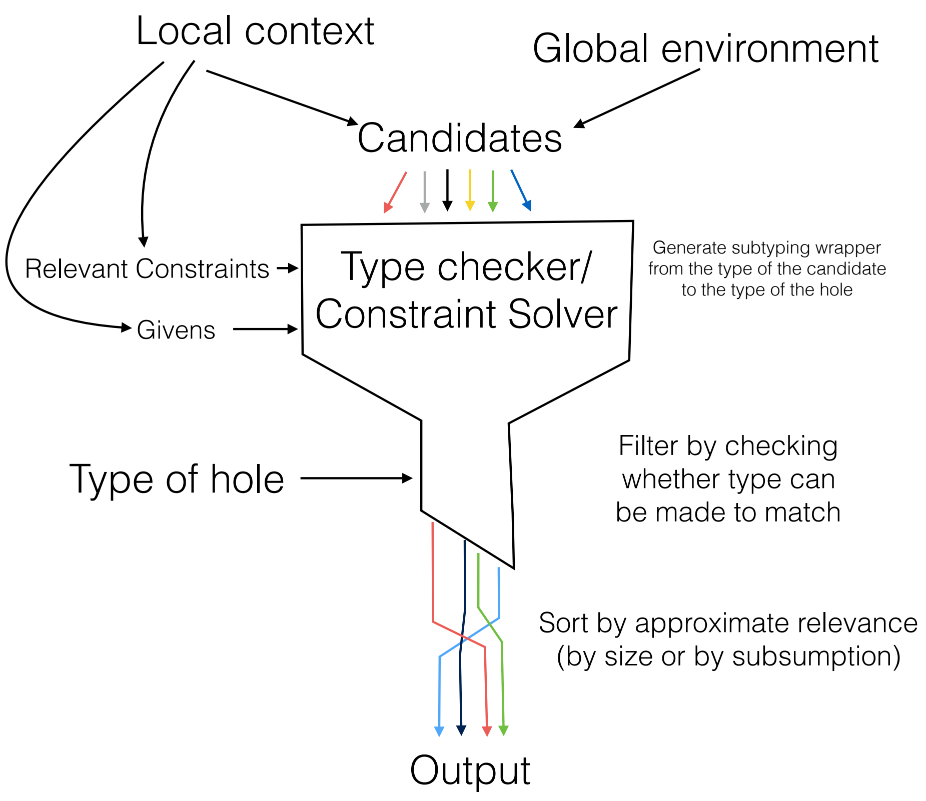

Local context
Global environment
Candidates
Relevant Constraints
Givens
Type checker/
Constraint Solver
Generate subtyping wrapper from the type of the candidate to the type of the hole
Type of hole
Filter by checking whether type can be made to match
Sort by approximate relevance (by size or by subsumption)
Output

Figure 3.1: Finding valid hole fits

3.1.1 The Inputs

When reporting a typed-hole error, we generate the valid hole fits message by invoking the `findValidHoleFits` function. This function is passed 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 implications that these wanted constraints are nested within. We also pass along the tidy type environment of the error reporter.

This tidy type environment is used when we zonk the types and type variables before we output them to the user (see section 2.8 on zonking). By maintaining a consistent environment throughout the error reporting process, we ensure consistency in that if a type-variable is reported twice, it has the same representation in the output. We return a (possibly) updated type environment along with the generated message, to ensure that any later error reports have consistent variable naming as well.

A hole is represented with a `CHoleExpr` constraint. This constraint contains information about the hole such as its type, its type checker level, its location, and the local bindings of the hole, which contain identifiers such as the arguments to the function the hole is contained in and any identifiers bound in a where or let clause.

The simple constraints are constraints that were not solved during simplification, most likely due to their solution being dependent on some type variable bound by the type of the hole. From these simple constraints, we find “relevant constraints”,

which are constraints that mention any of the free type variables in the hole. Since there might be two holes that each have their own free variables, we filter out the passed simple constraints to only include any relevant constraints, which might be solved by filling in the hole.
These constraints may include type class constraints such as `Show a` or `Num a`, where `a` is the type of the hole. By taking care to include these unsolved constraints, we make sure that we do not suggest any hole fits that might be invalid due to not fulfilling the constraints on the type of the hole.
The implications that the hole is nested in contain information about the bound skolems such as any givens. This means that if we have a function `f`, where `f :: Show a => a -> String` and a definition, `f x = show _`, then these implications contain the information that `a` fulfills the constraint `Show a`, and thus we can suggest `x :: a` as a valid fit for the hole.
The `findValidHoleFits` function is run in the type checker monad, `TcM`, which carries with it a lot of the context that `findValidHoleFits` needs and some implicit inputs. The most important of these are the global reader elements, which are the identifiers that are in scope in the current context. They are the identifiers that are defined either locally in the module or in scope via an import. Other implicit inputs include the flags used to control the sorting and display of the found fits. Additional implicit inputs, such as what language extensions are enabled, come into play when we invoke the simplifier, but they are not explicitly handled otherwise.
We take the global reader elements and the local bindings, and we remove any bindings which are shadowed by more local bindings. We then pass these candidates to the `findSubs` function, which runs the main “loop” of the process and iterates through the possible candidates and checks whether they are a match.

3.1.2 Checking for a Match

When seeing whether an element is a match, we start by looking it up to find its identifier. For local bindings, we can just return the identifier that we retrieved when we got the local bindings from the hole. For global elements, we have to look the element up in the type checker monad, and find the identifier if it is either an `Id` or a data constructor, or throw it away if it is something other than an identifier, such as a type constructor or a type coercion. We then pass the type of the identifier to the fit checker.
The fit checker is wrapped with the `withoutUnification` wrapper, which takes the list of free variables in the types that we are to check, and makes sure (by copying their mutable parts) that any side-effects that the hole fit checker might have on these variables is reverted after the check. We also have to make sure that we do not affect the implications that the hole is wrapped in during the check, so we make sure to replace the `EvBindsVar` that they contain with a fresh one, so that any evidence gathered during the check is discarded afterwards. We then pass this to the `tcCheckHoleFit` function.
The `tcCheckHoleFit` function encapsulates our interaction with the simplifier. There, we start by creating a subtype wrapper with `tcSubType` and capturing any constraints generated. We then add the relevant constraints to these constraints, and

pass this to the simplifier. If the simplifier says that it managed to simplify and solve all the constraints, we accept the fit and return the subtype wrapper, otherwise we reject it.

3.1.3 Gathering the Output

If the `tcCheckHoleFit` function reports a match, we take the returned `HsWrapper`, unwrap any type applications and gather the list of types that the type would need to be applied to. These are the "type application" arguments that we display in the output. These applications might refer to unified type variables, so we zonk the types in the wrapper before outputting.

We then wrap the fits in the `HoleFit` type, which contains the element, the id, the type, the refinement level (the number of additional holes) and the type application wrapper types. These are then returned as the valid hole fits from the `findSubs` function.

We then take these fits and tidy them by zonking them with the tidy type environment. This means that we resolve all type variables if possible, and have consistent names for these type variables in the output, e.g. if the type variable a appears in the output, then any type variable called a in the valid hole fits output will be that same type variable.

3.1.4 Sorting the Output

After we have found the valid hole fits and tidied them, we sort them so that the most relevant fits appear before less relevant fits. As an approximation for relevance, we use the size of the types in the wrapper needed for the type applications. This means that if we would need to instantiate the identifier at many large (i.e. complex) types, it is less relevant than if we need only to instantiate it at few simple types. The most relevant fit is then the one where we do not need to instantiate at all, which means that it is an exact fit.

An alternative sorting method of sorting by subsumption is available via a flag. Subsumption is when one type is more or equally general than another type [7]. This sorting method sorts by creating a subsumption graph of each of the fits, i.e. checks for each fit which (if any) other fits it is subsumed by. The valid hole fits are then sorted by a graph sort, so that more general fits (i.e. those that subsume other fits) appear after less general fits. This, however, is much more time consuming, so we do not use this method unless the flag is set. As an example, the subsumption graph for `(_ "hello, world") :: [String]` is:

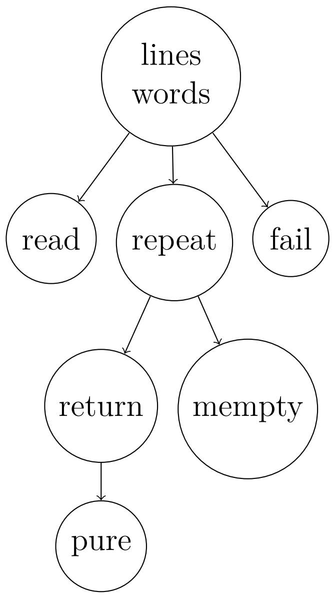


Figure 3.2: Subsumption graph of the hole fits in `(_ "hello, world") :: [String]`

The output will then be as follows, shown here compared with the output when sorting by size (the default):

```
Valid hole fits include
  lines :: String -> [String]
  words :: String -> [String]
  read :: Read a => String -> a
    with read @[String]
  repeat :: a -> [a]
    with repeat @String
  mempty :: Monoid a => a
    with mempty @([Char] -> [String])
  return :: Monad m => a -> m a
    with return @[] @String
  pure :: Applicative f => a -> f a
    with pure @[] @String
  fail :: Monad m => String -> m a
    with fail @[] @String
```

(a) Sorting by subsumption.

```
Valid hole fits include
  lines :: String -> [String]
  words :: String -> [String]
  repeat :: forall a. a -> [a]
    with repeat @String
  fail :: Monad m => String -> m a
    with fail @[] @String
  return :: Monad m => a -> m a
    with return @[] @String
  pure :: Applicative f => a -> f a
    with pure @[] @String
  read :: Read a => String -> a
    with read @[String]
  mempty :: Monoid a => a
    with mempty @([Char] -> [String])
```

(b) Sorting by size (the default).

Figure 3.3: The valid hole fits of `(_ "hello, world") :: [String]` when sorting by subsumption vs. sorting by size (`forall` quantifiers removed for clarity).

3.2 Finding Refinement Hole Fits

For refinement hole fits, we do very much the same as before. However, to simulate the additional holes, we check whether a slightly different type is a fit. Instead of looking for fits of the type of the hole, we check for fits of the type v `->` t, where t

is the type of the hole and v is a fresh type variable. To check for fits with multiple holes in them, we check for a fit for the type v_1 `->` $\cdots$ `->` v_n `->` t, where $v_1, \ldots, v_n$ are all fresh type variables. This allows us to suggest functions whose type would match the type of the hole if they were passed arguments of the right type.
We create these fresh type variables by creating a new open type kind and a new flexible type variable, which we create at the level of the hole. We then wrap the type of the hole by making a function type with these variables as type variable types.
We then run the check as we did before for valid hole fits, but using the newly wrapped type of the hole in place of the original type of the hole. When we find a fit, we have to take additional care to make sure that these fits are concrete, i.e. that all the fresh type variables got unified with some type. Otherwise, we would get matches like `id (_ :: a0 -> t) (_ :: a0)`, where `a0` can be any type at all. However, when dealing with complex types such as the free monad, matches like these can be useful, like in the case of `fmap (_ :: a0 -> Free f b) (_ :: f a0)`. We therefore allow the user to specify that matches like these are desirable with the flag `-fabstract-refinement-hole-fits`.
Additionally, when we find a fit, we zonk the fresh type variables and find which type they ended up being unified with. This allows us to display the type that these new holes must have in order for the refinement to be a valid hole fit.
The sorting and output of these refinement hole fits is the same as for valid hole fits, except that we output the additional holes and their types as well.

4

Evaluation

The goal of this thesis is to show how type systems can be used to go beyond verification to improve the user experience, and to extend the typed-holes of GHC to facilitate type-driven development in Haskell. To determine whether I have achieved this goal, I will evaluate the valid hole fits on a few practical examples in addition to the motivating examples we saw earlier.

4.1 A Note on Performance

When implementing the valid hole fits, performance was not my primary concern, since any performance hit would only be taken when the code being compiled contains a typed-hole, which is an error. However, care was taken to make sure that the performance was not too slow for practical use. As such, the valid hole fits perform reasonably well, with speeds that make interactive use bearable when the default flags are used. For most of the following examples, the runtime is $0.1 - 0.3$ seconds, with higher levels of `-frefinement-level-hole-fits` taking longer.
There are some degenerate cases where finding the valid hole fits can take a long time, such as when the type of the hole is an unconstrained type variable, since this makes it so that everything in scope is a valid hole fit. This degrades performance considerably when using the `-fsort-by-subsumption-hole-fits` flag, since it forces us to sort the entire prelude by subsumption, which can take upwards of 5 minutes. However, when using the default sorting method of sorting by size, even this degenerate case takes only 0.8 seconds.

4.2 Evaluation of Motivating Examples

4.2.1 Working with the Prelude

In our "Working with the Prelude" motivating example, we considered the following code.

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

For which GHC 8.2.1 reports the error message:

```
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)
```

Recall that when we looked up the type `_ :: [Char] -> [String]` in Hoogle, none of the results were from the Prelude, and only one of them had the exact type. However, when we compile this with GHC with my extension, we now get:

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

which includes the `lines` and `words` functions that we expected to find in Hoogle, but would not be found unless we looked for `String -> [String]`. The rest of the suggestions are not that useful, but they match the expected type and the user is shown how the function is instantiated, which can help newer users learn the prelude.

4.2.2 The Lens Library

For the `lens` library, we used as a motivating example the following code:

```haskell
module LensDemo where

import Control.Lens
import Control.Monad.State

newtype Test = Test { _v :: Int } deriving (Show)

val :: Lens' Test Int
val f (Test i) = Test <$> f i

updTest :: Test -> Test
updTest t = t &~ do
    _ val (1 :: Int)
```

Recall that when compiled with GHC 8.2.1, we got the following message:

```
Found hole:
   _ :: ((Int -> f0 Int) -> Test -> f0 Test)
        -> Int -> State Test a0
 Where: ‘f0’ is an ambiguous type variable
        ‘a0’ is an ambiguous type variable
 ...
```

When compiled with the new version, we now get the same message, but we additionally get the following:

```
Valid hole fits include
  (#=) :: forall s (m :: * -> *) a b. MonadState s m =>
          ALens s s a b -> b -> m ()
    with (#=) @Test @(StateT Test Identity) @Int @Int
  (<#=) :: forall s (m :: * -> *) a b. MonadState s m =>
           ALens s s a b -> b -> m b
    with (<#=) @Test @(StateT Test Identity) @Int @Int
  (<*=) :: forall s (m :: * -> *) a. (MonadState s m, Num a) =>
           LensLike’ ((,) a) s a -> a -> m a
    with (<*=) @Test @(StateT Test Identity) @Int
  (<+=) :: forall s (m :: * -> *) a.
           (MonadState s m, Num a) =>
           LensLike’ ((,) a) s a -> a -> m a
    with (<+=) @Test @(StateT Test Identity) @Int
  (<-=) :: forall s (m :: * -> *) a.
           (MonadState s m, Num a) =>
           LensLike’ ((,) a) s a -> a -> m a
    with (<-=) @Test @(StateT Test Identity) @Int
  (<<*=) :: forall s (m :: * -> *) a.
            (MonadState s m, Num a) =>
            LensLike’ ((,) a) s a -> a -> m a
    with (<<*=) @Test @(StateT Test Identity) @Int
  (Some hole fits suppressed;
   use -fmax-valid-hole-fits=N or -fno-max-valid-hole-fits)
```

These are all functions from within the `lens` library, relevant to the context that we are in, and likely to be functions that we might be looking for!
Here, we have a clear advantage by plugging in to the type checker machinery of GHC itself, and have more information available than we can easily pass to Hoogle. This means that we can suggest matches such as `(#=)`, `(<#=)` and `(<*=)` which fit the hole, even though their type is very far from the type of the hole lexicographically.

4.3 Exercises from Programming in Haskell

To evaluate the usefulness of valid hole fits for beginners, we will use a typed-hole-driven approach to solve a few select exercises from Graham Hutton’s introductory text, Programming in Haskell [17]. We will evaluate whether the suggestions listed contain the desired function, how early it is displayed in the list of suggestions

and what steps must be taken for the typed-hole suggestions to be useful, if at all possible.
To show the full power of the valid hole fits, I assume for these exercises that the user has set the more useful flags for beginners, which are:

- `-frefinement-level-hole-fits=N` with $N = 1$ or $N = 2$ to include the refinement fits, and
- `-fshow-type-app-vars-of-hole-fits`, which uses a variable based approach to displaying what type application would be used.

These flags are not the default in GHC at the moment. In the case of `-frefinement-level-hole-fits=N` there is a lot of output and it is hard to choose a sensible default for N, and in the case of `-fshow-type-app-vars-of-hole-fits` the syntax used is less intuitive to those familiar with the `TypeApplications` extension. However, these flags may become the default later, if well received by the community. For beginners, these flags can be set in either the source file (provided by an instructor for students to fill in), or in a configuration file for GHC or GHCi.

4.3.1 Exercises

- In exercise 1.7.3, students are asked to implement the `product` function for multiplying the elements of a list. Let us see how that works with the typed-holes. We begin by copying the `sum` example earlier in the book, but replacing $*$ with $+$:

```
myProduct :: [Int] -> Int
myProduct [] = 1
myProduct (n:ns) = _a * _b
```

  When we compile this piece of code, the list of valid hole fits for both `_a` and `_b` (since they are the same type) are:

```
Valid hole fits include
  n :: Int (bound at ex.hs:5:12)
  maxBound :: forall a. Bounded a => a
    with a ~ Int
    (imported from ‘Prelude’ at ex.hs:1:8-9
      (and originally defined in ‘GHC.Enum’))
  minBound :: forall a. Bounded a => a
    with a ~ Int
    (imported from ‘Prelude’ at ex.hs:1:8-9
      (and originally defined in ‘GHC.Enum’))
Valid refinement hole fits include
  myproduct (_ :: [Int])
    where myproduct :: [Int] -> Int
    (bound at ex.hs:4:1)
  ...
```

  Here we see both the identifiers used in the correct solution, `n` and `myproduct (_ :: [Int])` are at the top of the list of suggestions in each list respectively.

So the correct implementation, namely

```
myProduct :: [Int] -> Int
myProduct [] = []
myProduct (n:ns) = n * (myproduct ns)
```

can be found by trying the candidates each in turn until the example given, `product [2,3,4] = 24`, gives the correct output.
We see from this example that the typed-hole suggestions can be useful, and as noted, the desired identifiers are at the top of the lists of suggestions. The user still has to choose between the correct identifiers, and identifiers like `minBound` and `maxBound` might not be very relevant to those just starting out, even though they are valid hole fits. As noted in the future work section, using dependent types would improve this situation, since then the expected output could be written as an annotation as is done in Liquid Haskell [26].

- In exercise 4.8.1, students are asked to implement `halve :: [a] -> ([a], [a])`, a function which splits an even-lengthed list into two halves. When they compile the following code:

```
halve :: [a] -> ([a], [a])
halve = _
```

the output is:

```
Found hole: _ :: [a] -> ([a], [a])
Where: ‘a’ is a rigid type variable bound by
        the type signature for:
          halve :: forall a. [a] -> ([a], [a])
        at ex.hs:6:1-25
In the expression: _
In an equation for ‘halve’: halve = _
Relevant bindings include
  halve :: [a] -> ([a], [a]) (bound at ex.hs:7:1)
Valid hole fits include
  halve :: [a] -> ([a], [a])
  return :: forall (m :: * -> *). Monad m => forall a. a -> m a
    with m ~ ((,) [a]), a ~ [a]
  pure :: forall (f :: * -> *). Applicative f => forall a. a -> f a
    with f ~ ((,) [a]), a ~ [a]
  mempty :: forall a. Monoid a => a
    with a ~ ([a] -> ([a], [a]))
Valid refinement hole fits include
  break (_ :: a -> Bool)
    where break :: forall a. (a -> Bool) -> [a] -> ([a], [a])
    with a ~ a
  span (_ :: a -> Bool)
    where span :: forall a. (a -> Bool) -> [a] -> ([a], [a])
    with a ~ a
  splitAt (_ :: Int)
    where splitAt :: forall a. Int -> [a] -> ([a], [a])
    with a ~ a
...
```

Here, the `splitAt` function is the only function that mentions anything that is a number, like `Int`, and is indeed the correct solution. If you read the refinement as "splitAt an Int", it is very suggestive of what it does.

- In the first part of exercise 7.9.3, students are asked to redefine the `map f` function using `foldr`. We start by writing

```
mymap :: (a -> b) -> [a] -> [b]
mymap f = foldr _a _b
```

For the latter hole, we get:

```
Found hole: _b :: [b]
Where: 'b' is a rigid type variable bound by
        the type signature for:
          mymap :: forall a b. (a -> b) -> [a] -> [b]
        at hutton.hs:3:1-31
Or perhaps '_b' is mis-spelled, or not in scope
In the second argument of 'foldr', namely '_b'
In the expression: foldr _a _b
In an equation for 'mymap': mymap f = foldr _a _b
Relevant bindings include
  f :: a -> b (bound at hutton.hs:4:7)
  mymap :: (a -> b) -> [a] -> [b] (bound at hutton.hs:4:1)
Valid hole fits include
  mempty :: forall a. Monoid a => a
  with a ~ [b]
```

so we can replace `_b` with `mempty`.

Let us now try to work out the first hole:

```
mymap :: (a -> b) -> [a] -> [b]
mymap f = foldr _ mempty
```

When compiled, we get:

```
Found hole: _ :: a -> [b] -> [b]
Where: 'a', 'b' are rigid type variables bound by
        the type signature for:
          mymap :: forall a b. (a -> b) -> [a] -> [b]
        at hutton.hs:4:1-31
In the first argument of 'foldr', namely '_'
In the expression: foldr _ mempty
In an equation for 'mymap': mymap f = foldr _ mempty
Relevant bindings include
  f :: a -> b (bound at hutton.hs:5:7)
  mymap :: (a -> b) -> [a] -> [b] (bound at hutton.hs:5:1)
Valid hole fits include
  seq :: forall a b. a -> b -> b
    with a ~ a, b ~ [b]
  mempty :: forall a. Monoid a => a
    with a ~ (a -> [b] -> [b])
```

but none of these matches seem useful. Let us try binding the first argument:

```
mymap :: (a -> b) -> [a] -> [b]
mymap f = foldr k mempty
        where k x = _
```

When we compile the above, we now get:

```
Found hole: _ :: t
Where: ‘t’ is a rigid type variable bound by
          the inferred type of k :: p -> t
          at hutton.hs:5:9-15
In the expression: _
In an equation for ‘k’: k x = _
In an equation for ‘mymap’:
    mymap f
      = foldr k mempty
      where
          k x = _
Relevant bindings include
  x :: p (bound at hutton.hs:5:11)
  k :: p -> t (bound at hutton.hs:5:9)
  f :: a -> b (bound at hutton.hs:4:7)
  mymap :: (a -> b) -> [a] -> [b] (bound at hutton.hs:4:1)
```

Here, we hit a snag due to GHC inferring a too general type for `k` to be useful. To get any further, we either have to resort to `-XScopedTypeVariables` to give `k` the signature `a -> [b] -> [b]`, or we can bind the variable with a lambda expression. Let us go the latter route, since newcomers will likely be uncomfortable with adding extra extensions.

We write:

```
mymap :: (a -> b) -> [a] -> [b]
mymap f = foldr (\x -> _) mempty
```

and when compiled, this gives us:

```
Found hole: _ :: [b] -> [b]
Where: ‘b’ is a rigid type variable bound by
        the type signature for:
          mymap :: forall a b. (a -> b) -> [a] -> [b]
        at hutton.hs:3:1-31
In the expression: _
In the first argument of ‘foldr’, namely ‘(\ x -> _)’
In the expression: foldr (\ a -> _) mempty
Relevant bindings include
  x :: a (bound at hutton.hs:4:19)
  f :: a -> b (bound at hutton.hs:4:7)
  mymap :: (a -> b) -> [a] -> [b] (bound at hutton.hs:4:1)
Valid hole fits include
  cycle :: forall a. [a] -> [a]
    with a ~ b
  init :: forall a. [a] -> [a]
    with a ~ b
  reverse :: forall a. [a] -> [a]
    with a ~ b
  tail :: forall a. [a] -> [a]
    with a ~ b
  id :: forall a. a -> a
    with a ~ [b]
  mempty :: forall a. Monoid a => a
    with a ~ ([b] -> [b])
Valid refinement hole fits include
  (++) (_ :: [b])
    where (++) :: forall a. [a] -> [a] -> [a]
    with a ~ b
  filter (_ :: b -> Bool)
    where filter :: forall a. (a -> Bool) -> [a] -> [a]
    with a ~ b
  map (_ :: b -> b)
    where map :: forall a b. (a -> b) -> [a] -> [b]
    with a ~ b, b ~ b
  drop (_ :: Int)
    where drop :: forall a. Int -> [a] -> [a]
    with a ~ b
  dropWhile (_ :: b -> Bool)
    where dropWhile :: forall a. (a -> Bool) -> [a] -> [a]
    with a ~ b
  scanl1 (_ :: b -> b -> b)
    where scanl1 :: forall a. (a -> a -> a) -> [a] -> [a]
    with a ~ b
  (Some refinement hole fits suppressed;
    use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)
```

Now we are getting somewhere. We see that none of the valid hole fits would do what we want to do here, so let us keep going with the refinement hole fit `(++) _` by writing:

```
mymap :: (a -> b) -> [a] -> [b]
mymap f = foldr (\x -> (++) _ ) mempty
```

When we compile that, we get:

```
Found hole: _ :: [b]
Where: ‘b’ is a rigid type variable bound by
        the type signature for:
          mymap :: forall a b. (a -> b) -> [a] -> [b]
        at hutton.hs:3:1-31
In the first argument of ‘(++)’, namely ‘_’
In the expression: (++) _
In the first argument of ‘foldr’, namely ‘(\ a -> (++) _)’
Relevant bindings include
  x :: a (bound at hutton.hs:4:19)
  f :: a -> b (bound at hutton.hs:4:7)
  mymap :: (a -> b) -> [a] -> [b] (bound at hutton.hs:4:1)
Valid hole fits include
  mempty :: forall a. Monoid a => a
    with a ~ [b]
Valid refinement hole fits include
  cycle (_ :: [b])
    where cycle :: forall a. [a] -> [a]
    with a ~ b
  init (_ :: [b])
    where init :: forall a. [a] -> [a]
    with a ~ b
  repeat (_ :: b)
    where repeat :: forall a. a -> [a]
    with a ~ b
  reverse (_ :: [b])
    where reverse :: forall a. [a] -> [a]
    with a ~ b
  tail (_ :: [b])
    where tail :: forall a. [a] -> [a]
    with a ~ b
  fail (_ :: String)
    where fail :: forall (m :: * -> *).
                  Monad m =>
                  forall a. String -> m a
    with m ~ [], a ~ b
  (Some refinement hole fits suppressed;
    use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)
```

No clear path forward presents itself, so let us think for a bit. We need a `[b]` according to the hole, and we have both `x :: a` and `f :: a -> b`. Let us try writing:

```
mymap :: (a -> b) -> [a] -> [b]
mymap f = foldr (\x -> (++) [_]) mempty
```

When we compile this, we get:

```
Found hole: _ :: b
Where: ‘b’ is a rigid type variable bound by
         the type signature for:
           mymap :: forall a b. (a -> b) -> [a] -> [b]
         at hutton.hs:2:1-31
In the expression: _
In the first argument of ‘(++)’, namely ‘[_]’
In the expression: (++) [_]
Relevant bindings include
  x :: a (bound at hutton.hs:3:19)
  f :: a -> b (bound at hutton.hs:3:7)
  mymap :: (a -> b) -> [a] -> [b] (bound at hutton.hs:3:1)
Valid refinement hole fits include
  f (_ :: a)
    where f :: a -> b
```

Let us use the suggested `f` and write:

```haskell
mymap :: (a -> b) -> [a] -> [b]
mymap f = foldr (\x -> (++) [f _]) mempty
```

When compiled, this gives us:

```
Found hole: _ :: a
Where: ‘a’ is a rigid type variable bound by
         the type signature for:
           mymap :: forall a b. (a -> b) -> [a] -> [b]
         at hutton.hs:2:1-31
In the first argument of ‘f’, namely ‘_’
In the expression: f _
In the first argument of ‘(++)’, namely ‘[f _]’
Relevant bindings include
  x :: a (bound at hutton.hs:3:19)
  f :: a -> b (bound at hutton.hs:3:7)
  mymap :: (a -> b) -> [a] -> [b] (bound at hutton.hs:3:1)
Valid hole fits include x :: a
```

We can now finish our definition with:

```haskell
mymap :: (a -> b) -> [a] -> [b]
mymap f = foldr (\x -> (++) [f x]) mempty
```

Note that if it had suggested `(:) (_ :: b)`, the next refinement suggestion would have been `f (_ :: a)`, and we would end up with:

```haskell
mymap :: (a -> b) -> [a] -> [b]
mymap f = foldr (\x -> (:) (f x)) mempty
```

which is very concise. The suggestions do not consider `(:)` however, since it is built-in syntax and not an identifier in scope. The same goes for `\x -> _` and `[_]`, which is also built-in syntax and thus not considered. It would be nice to add these capabilities, and we have noted that in future work.

- In exercise 12.5.4, students are asked to define `Functor` and `Applicative` instances for `ZipList`, where `pure` should create an infinite list of copies of its argument, and <*> should apply each argument function to the corresponding argument value at the same position. The exercises provides the following skeleton to fill in:

```
newtype ZipList a = Z [a] deriving Show

instance Functor ZipList where
  -- fmap :: (a -> b) -> ZipList [a] -> ZipList [b]
  fmap g (Z xs) = undefined

instance Applicative ZipList where
  -- pure :: a -> ZipList [a]
  pure x = undefined

  -- <*> :: ZipList (a -> b) -> ZipList [a] -> ZipList [b]
  (Z gs) <*> (Z xs) = undefined
```

We begin by writing:

```
instance Functor ZipList where
  -- fmap :: (a -> b) -> ZipList [a] -> ZipList [b]
  fmap g (Z xs) = _
```

and when we compile, we get:

```
Found hole: _ :: ZipList b
Where: ‘b’ is a rigid type variable bound by
         the type signature for:
           fmap :: forall a b. (a -> b) -> ZipList a -> ZipList b
         at hutton.hs:7:4-7
In the expression: _
In an equation for ‘fmap’: fmap g (Z xs) = _
In the instance declaration for ‘Functor ZipList’
Relevant bindings include
  xs :: [a] (bound at hutton.hs:7:14)
  g :: a -> b (bound at hutton.hs:7:9)
  fmap :: (a -> b) -> ZipList a -> ZipList b (bound at hutton.hs:7:4)
Valid refinement hole fits include
  Z (_ :: [b])
    where Z :: forall a. [a] -> ZipList a
    with a ~ b
  ...
```

We use `Z _` and continue by writing:

```
instance Functor ZipList where
  -- fmap :: (a -> b) -> ZipList [a] -> ZipList [b]
  fmap g (Z xs) = Z _
```

But the list of suggestions does not have anything that contains the type `a`. However, we know that we need to use `a` somewhere in the definition, and that we want to do something with `xs`, so let us try the following:

```
instance Functor ZipList where
  -- fmap :: (a -> b) -> ZipList [a] -> ZipList [b]
  fmap g (Z xs) = Z (_ xs)
```

When we compile the above, we get:

```
Found hole: _ :: [a] -> [b]
Where: ‘a’, ‘b’ are rigid type variables bound by
         the type signature for:
           fmap :: forall a b. (a -> b) -> ZipList a -> ZipList b
         at hutton.hs:7:4-7
In the expression: _
In the first argument of ‘Z’, namely ‘(_ xs)’
In the expression: Z (_ xs)
Relevant bindings include
  xs :: [a] (bound at hutton.hs:7:14)
  g :: a -> b (bound at hutton.hs:7:9)
  fmap :: (a -> b) -> ZipList a -> ZipList b (bound at hutton.hs:7:4)
Valid hole fits include
  mempty :: forall a. Monoid a => a
    with a ~ ([a] -> [b])
Valid refinement hole fits include
  map (_ :: a -> b)
    where map :: forall a b. (a -> b) -> [a] -> [b]
  ...
```

Since we have `g :: a -> b` we choose `map`, and finish the definition by writing:

```
instance Functor ZipList where
  -- fmap :: (a -> b) -> ZipList [a] -> ZipList [b]
  fmap g (Z xs) = Z (map g xs)
```

For the second part of the exercise, we write:

```
instance Applicative ZipList where
  -- pure :: a -> ZipList [a]
  pure x = _
```

and when we compile, we get:

```
Found hole: _ :: ZipList a
Where: ‘a’ is a rigid type variable bound by
         the type signature for:
           pure :: forall a. a -> ZipList a
         at hutton.hs:11:3-6
In the expression: _
In an equation for ‘pure’: pure x = _
In the instance declaration for ‘Applicative ZipList’
Relevant bindings include
  x :: a (bound at hutton.hs:11:8)
  pure :: a -> ZipList a (bound at hutton.hs:11:3)
Valid refinement hole fits include
  pure (_ :: a)
    where pure :: a -> ZipList a
  Z (_ :: [a])
    where Z :: forall a. [a] -> ZipList a
    with a ~ a
  pure (_ :: a)
    where pure :: forall (f :: * -> *).
                  Applicative f =>
                  forall a. a -> f a
    with f ~ ZipList, a ~ a
  id (_ :: ZipList a)
    where id :: forall a. a -> a
    with a ~ (ZipList a)
  head (_ :: [ZipList a])
    where head :: forall a. [a] -> a
    with a ~ (ZipList a)
  last (_ :: [ZipList a])
    where last :: forall a. [a] -> a
    with a ~ (ZipList a)
  (Some refinement hole fits suppressed;
   use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)
```

We know that we want to return an infinite list of copies and that we can not use the `pure` function since that is the function we are defining, so we go with `Z _` and write:

```haskell
instance Applicative ZipList where
  -- pure :: a -> ZipList [a]
  pure x = Z _
```

which, when compiled, gives us:

```
Found hole: _ :: [a]
Where: ‘a’ is a rigid type variable bound by
         the type signature for:
           pure :: forall a. a -> ZipList a
         at hutton.hs:11:3-6
In the first argument of ‘Z’, namely ‘_’
In the expression: Z _
In an equation for ‘pure’: pure x = Z _
Relevant bindings include
  x :: a (bound at hutton.hs:11:8)
  pure :: a -> ZipList a (bound at hutton.hs:11:3)
Valid hole fits include
  mempty :: forall a. Monoid a => a
    with a ~ [a]
Valid refinement hole fits include
  cycle (_ :: [a])
    where cycle :: forall a. [a] -> [a]
    with a ~ a
  init (_ :: [a])
    where init :: forall a. [a] -> [a]
    with a ~ a
  repeat (_ :: a)
    where repeat :: forall a. a -> [a]
    with a ~ a
  ...
```

We want `pure` to return an infinite list, which is what `repeat` does, and we have `x :: a` in the relevant bindings, so we can finish the definition with:

```haskell
instance Applicative ZipList where
  -- pure :: a -> ZipList a
  pure x = Z (repeat x)
```

Now for the last part of the second exercise. Within the `Applicative` instance definition for `ZipList`, we write:

```haskell
-- <*> :: ZipList (a -> b) -> ZipList a -> ZipList b
(Z gs) <*> (Z xs) = _
```

and when we compile, we get:

```
Found hole: _ :: ZipList b
Where: ‘b’ is a rigid type variable bound by
         the type signature for:
           (<*>) :: forall a b. ZipList (a -> b) -> ZipList a -> ZipList b
         at hutton.hs:14:10-12
In the expression: _
In an equation for ‘<*>’: (Z gs) <*> (Z xs) = _
In the instance declaration for ‘Applicative ZipList’
Relevant bindings include
  xs :: [a] (bound at hutton.hs:14:17)
  gs :: [a -> b] (bound at hutton.hs:14:6)
  (<*>) :: ZipList (a -> b) -> ZipList a -> ZipList b
    (bound at hutton.hs:14:10)
Valid refinement hole fits include
  Z (_ :: [b])
    where Z :: forall a. [a] -> ZipList a
    with a ~ b
  (<*>) (_ :: ZipList (a -> b)) (_ :: ZipList a)
    where (<*>) :: ZipList (a -> b) -> ZipList a -> ZipList b
  pure (_ :: b)
    where pure :: forall (f :: * -> *).
                  Applicative f =>
                  forall a. a -> f a
    with f ~ ZipList, a ~ b
  id (_ :: ZipList b)
    where id :: forall a. a -> a
    with a ~ (ZipList b)
  head (_ :: [ZipList b])
    where head :: forall a. [a] -> a
    with a ~ (ZipList b)
  last (_ :: [ZipList b])
    where last :: forall a. [a] -> a
    with a ~ (ZipList b)
  (Some refinement hole fits suppressed;
   use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)
```

We want to return a ZipList and know how to handle `[b]`, so we go with `Z _` and write:

```
-- <*> :: ZipList (a -> b) -> ZipList a -> ZipList b
(Z gs) <*> (Z xs) = Z  _
```

which, as we encountered when defining `pure`, does not output anything that contains the type `a`. We do know that we want to use the `xs`, so we write:

```
-- <*> :: ZipList (a -> b) -> ZipList a -> ZipList b
(Z gs) <*> (Z xs) = Z  (_ xs)
```

which, when compiled, gives us:

```
Found hole: _ :: [a] -> [b]
Where: ‘a’, ‘b’ are rigid type variables bound by
         the type signature for:
           (<*>) :: forall a b. ZipList (a -> b)
                 -> ZipList a -> ZipList b
         at hutton.hs:14:10-12
In the expression: _
In the first argument of ‘Z’, namely ‘(_ xs)’
In the expression: Z (_ xs)
Relevant bindings include
  xs :: [a] (bound at hutton.hs:14:17)
  gs :: [a -> b] (bound at hutton.hs:14:6)
  (<*>) :: ZipList (a -> b) -> ZipList a -> ZipList b
    (bound at hutton.hs:14:10)
Valid hole fits include
  mempty :: forall a. Monoid a => a
    with a ~ ([a] -> [b])
Valid refinement hole fits include
  map (_ :: a -> b)
    where map :: forall a b. (a -> b) -> [a] -> [b]
    with a ~ a, b ~ b
  scanl (_ :: b -> a -> b) (_ :: b)
    where scanl :: forall b a. (b -> a -> b) -> b -> [a] -> [b]
    with b ~ b, a ~ a
  scanr (_ :: a -> b -> b) (_ :: b)
    where scanr :: forall a b. (a -> b -> b) -> b -> [a] -> [b]
    with a ~ a, b ~ b
  fmap (_ :: a -> b)
    where fmap :: forall (f :: * -> *).
                  Functor f =>
                  forall a b. (a -> b) -> f a -> f b
    with f ~ [], a ~ a, b ~ b
  (<*>) (_ :: [a -> b])
    where (<*>) :: forall (f :: * -> *).
                   Applicative f =>
                   forall a b. f (a -> b) -> f a -> f b
    with f ~ [], a ~ a, b ~ b
  concatMap (_ :: a -> [b])
    where concatMap :: forall (t :: * -> *) a b.
                       Foldable t =>
                       (a -> [b]) -> t a -> [b]
    with t ~ [], a ~ a, b ~ b
  (Some refinement hole fits suppressed;
   use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)
```

Here, `(<*>)` wants an argument of type `_ :: [a -> b]`, which we have from `gs` in the relevant bindings. We can now finish up the definition with:

```
(Z gs) <*> (Z xs) = Z ((<*>) gs xs)
```

But wait! This does not do what we want it to do!

Instead of applying each element of `gs` to the corresponding argument value at the same position in `xs`, it applies each element of `gs` to each element of `xs`, so instead of getting `[g1 x1, g2 x2, g3 x3]` that we want, we get `[g1 x1, g1 x2, g1 x3, g2 x1, g2 x2, g2 x3, g3 x1, g3 x2, g3 x3]`.

Let us try again, but this time, we know that we want to use `gs` and `xs`, so we write:

```
(Z gs) <*> (Z xs) = Z (_ gs xs)
```

which, when compiled, gives us:

```
Found hole: _ :: [a -> b] -> [a] -> [b]
Where: ‘a’, ‘b’ are rigid type variables bound by
         the type signature for:
           (<*>) :: forall a b. ZipList (a -> b)
                  -> ZipList a -> ZipList b
         at hutton.hs:14:10-12
In the expression: _
In the first argument of ‘Z’, namely ‘(_ gs xs)’
In the expression: Z (_ gs xs)
Relevant bindings include
  xs :: [a] (bound at hutton.hs:14:17)
  gs :: [a -> b] (bound at hutton.hs:14:6)
  (<*>) :: ZipList (a -> b) -> ZipList a -> ZipList b
    (bound at hutton.hs:14:10)
Valid hole fits include
  (<*>) :: forall (f :: * -> *).
           Applicative f =>
           forall a b. f (a -> b) -> f a -> f b
    with f ~ [], a ~ a, b ~ b
  mempty :: forall a. Monoid a => a
    with a ~ ([a -> b] -> [a] -> [b])
Valid refinement hole fits include
  zipWith (_ :: (a -> b) -> a -> b)
    where zipWith :: forall a b c. (a -> b -> c) -> [a] -> [b] -> [c]
    with a ~ (a -> b), b ~ a, c ~ b
...
```

we are implementing a *Zip*List, so let us choose `zipWith`. We write:

```
(Z gs) <*> (Z xs) = Z (zipWith _ gs xs)
```

which, when compiled, outputs:

```
Found hole: _ :: (a -> b) -> a -> b
Where: ‘a’, ‘b’ are rigid type variables bound by
         the type signature for:
           (<*>) :: forall a b. ZipList (a -> b)
                 -> ZipList a -> ZipList b
         at hutton.hs:14:10-12
In the first argument of ‘zipWith’, namely ‘_’
In the first argument of ‘Z’, namely ‘(zipWith _ gs xs)’
In the expression: Z (zipWith _ gs xs)
Relevant bindings include
  xs :: [a] (bound at hutton.hs:14:17)
  gs :: [a -> b] (bound at hutton.hs:14:6)
  (<*>) :: ZipList (a -> b) -> ZipList a -> ZipList b
    (bound at hutton.hs:14:10)
Valid hole fits include
  ($) :: forall a b. (a -> b) -> a -> b
    with r ~ 'GHC.Types.LiftedRep, a ~ a, b ~ b
  ($!) :: forall a b. (a -> b) -> a -> b
    with r ~ 'GHC.Types.LiftedRep, a ~ a, b ~ b
  id :: forall a. a -> a
    with a ~ (a -> b)
 ...
```

We want to *apply* the function, so we choose the application operator `($)`, and finish up with:

```
(Z gs) <*> (Z xs) = Z (zipWith ($) gs xs)
```

and we are done!

As we see, the valid hole fits work well for some of the exercises, but not as well for others. In the `mymap` example, we had to resort to lambda expressions to get any further, or we would have to enable `-fabstract-refinement-hole-fits` and try to work our way through using `(.)`, which does not really improve the user experience. In the same example, we also saw that the suggestions are not that useful when the correct answer uses built-in syntax, and not a function. This may not be too big an issue, as users are likely to learn the built-in syntax much earlier than they learn the functions in the Prelude.

We also saw that we had to do a lot of choosing, and our choices were based on what types were mentioned and what types we had available in the relevant bindings. As we saw in the `ZipList` example, the choices can sometimes be misleading, and even if the types match, the behavior might not be as intended.

We could improve the sorting of the valid hole fits by using this information, that is by ranking highest those functions whose arguments are any identifier in the relevant bindings.

4.4 The Free Monad

The free monad is an interesting concept from category theory [27, 28]. The free monad `Free` is a monad over a functor `f` for which there exists a transformation `s :: f a -> Free f a` that is universal in the sense that for every monad `m` over `f` and every transformation `s' :: f a -> m a` there exists a unique monad morphism `t :: Free f a -> m a` where `s' = t . s`.

It is an interesting test for the valid hole fits, since the types of the free monad and the operations we are defining are very descriptive. In Haskell, we can define the free monad like this:

```
data Free f a
  = Pure a
  | Free (f (Free f a))
```

Figure 4.1: The Free Monad

We can use the typed-holes to help us write the monad instance for the free monad. If we compile the following:

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

GHC will tell us that it has:

```
Found hole: _ :: Free f b
Where: ‘b’ is a rigid type variable bound by
         the type signature for:
           (>>=) :: forall a b. Free f a -> (a -> Free f b) -> Free f b
         at tfree.hs:9:10-12
       ‘f’ is a rigid type variable bound by
         the instance declaration
         at tfree.hs:7:10-36
In the expression: _
In an equation for ‘>>=’: Free f >>= g = _
In the instance declaration for ‘Monad (Free f)’
Relevant bindings include
  g :: a -> Free f b (bound at tfree.hs:10:14)
  f :: f (Free f a) (bound at tfree.hs:10:8)
  (>>=) :: Free f a -> (a -> Free f b) -> Free f b
    (bound at tfree.hs:9:10)
Constraints include Functor f (from tfree.hs:7:10-36)
Valid refinement hole fits include
  g (_ :: a)
    where g :: a -> Free f b
  Pure (_ :: b)
    with Pure @f @b
    where Pure :: forall (f :: * -> *) a. a -> Free f a
  Free (_ :: f (Free f b))
    with Free @f @b
    where Free :: forall (f :: * -> *) a. f (Free f a) -> Free f a
  (>>=) (_ :: Free f a) (_ :: a -> Free f b)
    where (>>=) :: Free f a -> (a -> Free f b) -> Free f b
  fail (_ :: String)
    with fail @(Free f) @b
    where fail :: forall (m :: * -> *). Monad m => forall a. String -> m a
  return (_ :: b)
    with return @(Free f) @b
    where return :: forall (m :: * -> *). Monad m => forall a. a -> m a
  (Some refinement hole fits suppressed;
    use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)
```

Indeed, the correct choice here is `Free (_ :: f (Free f b))`. Let us compile again with that hole:

```haskell
instance Functor f => Monad (Free f) where
  return a     = Pure a
  Pure a >>= f = f a
  Free f >>= g = Free _
```

We get (after enabling `-fabstract-refinement-hole-fits` to allow fresh type variables to appear in fits):

```
Valid refinement hole fits include
  fmap (_ :: a2 -> Free f b) (_ :: f a2)
    with fmap @f @a2 @(Free f b)
    where fmap :: forall (f :: * -> *).
                  Functor f =>
                  forall a b. (a -> b) -> f a -> f b
  (<$>) (_ :: a5 -> Free f b) (_ :: f a5)
    with (<$>) @f @a5 @(Free f b)
    where (<$>) :: forall (f :: * -> *) a b.
                   Functor f =>
                   (a -> b) -> f a -> f b
  (<$) (_ :: Free f b) (_ :: f b3)
    with (<$) @f @(Free f b) @b3
    where (<$) :: forall (f :: * -> *).
                  Functor f =>
                  forall a b. a -> f b -> f a
  id (_ :: t0 -> f (Free f b)) (_ :: t0)
    with id @(t0 -> f (Free f b))
    where id :: forall a. a -> a
  head (_ :: [t0 -> f (Free f b)]) (_ :: t0)
    with head @(t0 -> f (Free f b))
    where head :: forall a. [a] -> a
  last (_ :: [t0 -> f (Free f b)]) (_ :: t0)
    with last @(t0 -> f (Free f b))
    where last :: forall a. [a] -> a
  (Some refinement hole fits suppressed;
    use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)
```

Here the correct choice is `fmap _ _`, with `a2` ∼ `Free f a`. We fill in that refinement and continue:

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

For the latter hole, we get:

```
Valid hole fits include f :: f (Free f a)
```

We fill that in and look for refinements for:

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

When we compile the above, we get:

```
  Valid refinement hole fits include
    fmap (_ :: a -> b)
      with fmap @(Free f) @a @b
      where fmap :: forall (f :: * -> *). Functor f =>
                    forall a b. (a -> b) -> f a -> f b
    (<*>) (_ :: Free f (a -> b))
      with (<*>) @(Free f) @a @b
      where (<*>) :: forall (f :: * -> *). Applicative f =>
                     forall a b. f (a -> b) -> f a -> f b
    (<$>) (_ :: a -> b)
      with (<$>) @(Free f) @a @b
      where (<$>) :: forall (f :: * -> *) a b. Functor f =>
                     (a -> b) -> f a -> f b
    (=<<) (_ :: a -> Free f b)
      with (=<<) @(Free f) @a @b
      where (=<<) :: forall (m :: * -> *) a b. Monad m =>
                     (a -> m b) -> m a -> m b
    (<*) (_ :: Free f b)
      with (<*) @(Free f) @b @a
      where (<*) :: forall (f :: * -> *). Applicative f =>
                    forall a b. f a -> f b -> f a
    (<$) (_ :: b)
      with (<$) @(Free f) @b @a
      where (<$) :: forall (f :: * -> *). Functor f =>
                    forall a b. a -> f b -> f a
    (Some refinement hole fits suppressed;
      use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)
```

Here, we should choose `(=<<) (_ :: a -> Free f b)`, since we know from the relevant bindings that `g :: a -> Free f b`.
We now write:

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

which, when compiled, gives us:

```
Valid hole fits include g :: a -> (Free f b)
```

We then replace the hole with `g`:

```
instance Functor f => Monad (Free f) where
  return a     = Pure a
  Pure a >>= f = f a
  Free f >>= g = Free (fmap ((=<<) g) f)
```

and we have completed our definition!

As the evaluation shows, the valid hole fits can be of great use when dealing with category theoretical structures such as the free monad. The theory makes it so that we can write type signatures that match the domain accurately, and the valid hole fits always found the correct refinement, and displayed it early in the list of valid fits or refinements. We did have to enable some extra flags, notably the `-fabstract-refinement-hole-fits` so that it would find the correct match, but when we did it both found the match and displayed it at the top.
We also had to enable the `-XMonoLocalBinds` extension. `-XMonoLocalBinds` makes GHC infer less polymorphic types for local bindings by default [18]. This is good for typed-holes in general, since the less polymorphic the type is, the more accurate the valid hole fits can be. In this case, the inferred type of `(>>=)` is too polymorphic. Instead of

```
(>>=) :: forall a b. Free f a -> (a -> Free f b) -> Free f b
```

GHC infers the more polymorphic type of

```
(>>=) :: (Functor f1, Functor f2) => forall a b.
           Free f1 a -> (a -> Free f2 b) -> Free f2 b
```

and while correct in the case of `Free f >>= g = _`, this is not the type of `Free (fmap ((=<<) g) f)`, which we want in the end. Enabling `-XMonoLocalBinds` makes GHC infer the correct target type, which allows the full power of the valid hole fits to come into play.

4.5 Looking Up Functions with Non-Functional Properties

One use case for typed-holes is to interact with functions with non-functional properties. Non-functional properties are properties which are annotations to the types themselves. These annotations can include the security level of the computation, the computational complexity of the computation or even the power usage of the circuit that the function would generate. These non-functional properties are often expressed only in comments and documentation, but it is possible to wrap a type within a `newtype` and add these properties to the type system as explicit annotations.

4.5.1 Sorting Algorithms

Consider the following library of sorting functions, annotated with non-functional properties like computational complexity, memory complexity and stability (for the definition of the O notation annotation and full sorting library, see the source in section A.2.1 in the appendix):

```
newtype Sorted (cpu :: AsymptoticPoly) -- The minimum operational complexity
                                       -- this algorithm satisfies.
               (mem :: AsymptoticPoly) -- The minimum space complexity this
                                       -- algorithm satisfies.
               (stable :: Bool)        -- Whether the sort is stable or not.
               a                       -- What was being sorted.
               = Sorted {sortedBy :: [a]}

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

insertionSort :: (Ord a, n >=. O(N^.2), m >=. O(One), IsStable s)
              => [a] -> Sorted n m s a
insertionSort = ...

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

heapSort :: (Ord a, n >=. O(N*.LogN), m >=. O(One))
         => [a] -> Sorted n m False a
heapSort = ...
```

Figure 4.2: The Sorting library, for full implementation see section A.2.2 in the appendix

Since the functions are annotated with non-functional properties, the valid hole fits enables a search for functions with those properties. As shown by the definition of O notation type we can also find fits that satisfy more strict properties, that is with better complexity. If we need a stable sort with quadratic computational complexity and linear memory complexity for integers, we can search with the following:

```
{-# LANGUAGE TypeInType, TypeOperators #-}
module Main where

import Sorting
import ONotation

mySort :: Sorted (O(N^.2)) (O(N)) True Integer
mySort = _ [3,1,2]
```

And the valid hole fits will find those sorting functions defined in our library for which those non-functional properties hold:

```
 Valid hole fits include
   insertionSort :: forall a (n :: AsymptoticPoly) (m :: AsymptoticPoly)
                           (s :: Bool).
                           (Ord a, n >=. O (N ^. 2), m >=. O One
                           , Sorting.IsStable s) =>
                    [a] -> Sorted n m s a
   mergeSort :: forall a (n :: AsymptoticPoly) (m :: AsymptoticPoly)
                       (s :: Bool).
                       (Ord a, n >=. O (N *. LogN), m >=. O N
                       , Sorting.IsStable s) =>
                [a] -> Sorted n m s a
```

If we instead need $O(N \log(N))$ computational complexity but it can be unstable, we can search for that with:

```haskell
{-# LANGUAGE TypeInType, TypeOperators #-}
module Main where

import Sorting
import ONotation

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

Which results in:

```
 Valid hole fits include
   heapSort :: forall a (n :: AsymptoticPoly) (m :: AsymptoticPoly).
               (Ord a, n >=. O (N *. LogN), m >=. O One) =>
               [a] -> Sorted n m 'False a
   quickSort :: forall a (n :: AsymptoticPoly) (m :: AsymptoticPoly).
                (Ord a, n >=. O (N *. LogN), m >=. O N) =>
                [a] -> Sorted n m 'False a
   mergeSort :: forall a (n :: AsymptoticPoly) (m :: AsymptoticPoly)
                       (s :: Bool).
                       (Ord a, n >=. O (N *. LogN), m >=. O N
                       , Sorting.IsStable s) =>
                [a] -> Sorted n m s a
```

And finally, if we need constant memory complexity, we can search for that with:

```haskell
{-# LANGUAGE TypeInType, TypeOperators #-}
module Main where

import Sorting
import ONotation

mySort :: Sorted (O(N*.LogN)) (O(One)) False Integer
mySort = _ [3,1,2]
```

We will find that the output is:

```
  Valid hole fits include
    heapSort :: forall a (n :: AsymptoticPoly) (m :: AsymptoticPoly).
                (Ord a, n >=. O (N *. LogN), m >=. O One) =>
                [a] -> Sorted n m 'False a
```

As we can see, if we had a larger library of sorting algorithms or other types of functions such as for example adder circuits from Lava annotated with timing information, we could access the information embedded in the type annotation easily with the valid hole fits [29].

4.5.2 DCC

Consider a module using the Dependency Core Calculus (*DCC*), a small extension of the lambda calculus designed to secure *pure computation* using graded monads and type family lattices [30]. *DCC* allows us to encode the *privacy* of functions and values in Haskell. Privacy annotations can be used to verify that information is not being leaked to an attacker by returning private data in publicly accessible functions, unless explicitly acknowledged by the programmer.
For this evaluation, we use an implementation of *DCC* based on the paper "Encoding DCC in Haskell" [30], the source for which can be found in section A.3.1 in the appendix. The module assumes that we want to keep the name, location and age of the user private, but allows public access to some derived data, like whether the user is allowed to drink and what restaurants are nearby. In this example, we use `H` to denote high–security private information, and `L` to denote low–security public information.
Let us assume that a library author has authored a library for interacting with data about a user to display information about nearby restaurants. To ensure that data is not accidentally leaked by programmers using the library, the library authors have made use of DCC annotations to mark which functions can be used to get public data from the private information:

```
bestNearbyRestaurant :: T H User -> T L (Maybe Restaurant)
bestNearbyRestaurant = ...

isInGothenburg :: T H User -> T L Bool
isInGothenburg = ...

isAllowedToDrink :: T H User -> T L Bool
isAllowedToDrink = ...

readPublic :: T L a -> a
readPublic = ...

getUserInfo :: IO (T H User)
getUserInfo = ...
```

Figure 4.3: The Restaurant Info library (see section A.3.2 in the appendix for the full implementation)

Here the information about the user is private, and can only be obtained via the `getUserInfo` function, which annotates it with a security level of `H`, for high–security. To make use of this information, programmers can apply the functions included to read data derived from the private information that is allowed to be public, such as such as whether they are legally allowed to drink, whether they are in Gothenburg, and what restaurants are near the users location.
When a programmer would like to use this library to add a restaurant recommendation feature to their app, they might wonder what they can do with the private information that they access and how they can go from private to public. Then they could write:

```haskell
{-# LANGUAGE MonoLocalBinds #-}
import RestaurantInfo

main :: IO ()
main = do
  -- Get information about the user
  user <- getUserInfo
  -- What can I do with it?
  print $ readPublic $ _ user
```

And the valid hole fits will tell them that:

```
Valid hole fits include
  bestNearbyRestaurant :: T 'H User -> T 'L (Maybe Restaurant)
  isAllowedToDrink :: T 'H User -> T 'L Bool
  isInGothenburg :: T 'H User -> T 'L Bool
```

which are precisely the functions that the programmer can use to access the private information contained within the user data type.

4.6 The PiSkjerm Project

The PiSkjerm project is a hobby project to evaluate the usefulness of valid hole fits for typed-holes for real world Haskell. The project was based on using the API provided by the transportation authority for Gothenburg and the rest of Västra Götaland, “Västtraffik”, to display information about upcoming bus departures from nearby bus stops on a small eInk display on a Raspberry Pi. The full source is available in section A.4 in the appendix.
The project uses a simple translation of the DSL provided by the “InkyPhat” library by the screen vendor to interact with the screen. As the InkyPhat library is only available for Python, we interact with the screen by spawning a sub-process and sending commands and receiving responses via a file handle.
During the project, a lot of issues emerged with the cross-compilation tool-chain. For example, invoking the time functions in Haskell would cause a segmentation fault. We mitigate these flaws by using the Python interpreter to perform queries relating to time and the network (which uses the time library). Both versions are available in the source, and can be toggled by a build-time flag.

We will only look at how the valid hole fits helped when we were writing the initial part of the application, which fetches a token from Västtrafik, which is later used to authenticate requests to the API.

4.6.1 Fetching the Token

The Västtraffik API uses OAuth2 token authentication. which means that prior to requesting any information about departures, we first have to get a token to authenticate our API access. We get a “key” and a “secret” from the Västtraffik API when we register, and we need to use those to get the token.

4.6.1.1 Getting the Credentials

Since we do not want to check the key and secret into our code base, we pass them to the PiSkjerm application by environment variables “VASTTRAFIKKEY” and “VASTTRAFIKSECRET”. But how can we read those from the environment? Well, we start by importing `System.Environment`, and then we need to find a function that returns the values. We can ask GHC with the following code:

```
getAuthCredentials :: IO (String, String)
getAuthCredentials = do key <- _a "VASTTRAFIKKEY"
                        secret <- _b "VASTTRAFIKSECRET"
                        return (key, secret)
```

Here we have two holes, `_a` and `_b`, which both have types `IO String`. When we compile, we get:

```
Found hole: _b :: [Char] -> IO String
Or perhaps ‘_b’ is mis-spelled, or not in scope
In the expression: _b
In a stmt of a ’do’ block: secret <- _b "VASTTRAFIKSECRET"
In the expression:
  do key <- _a "VASTTRAFIKKEY"
     secret <- _b "VASTTRAFIKSECRET"
     return (key, secret)
Relevant bindings include
  key :: String (bound at skjerm.hs:5:25)
  getAuthCredentials :: IO (String, String) (bound at skjerm.hs:5:1)
Valid hole fits include
  readFile :: FilePath -> IO String
  getEnv :: String -> IO String
  readIO :: forall a. Read a => String -> IO a
    with readIO @String
  fail :: forall (m :: * -> *). Monad m => forall a. String -> m a
    with fail @IO @String
  return :: forall (m :: * -> *). Monad m => forall a. a -> m a
    with return @IO @String
  pure :: forall (f :: * -> *). Applicative f => forall a. a -> f a
    with pure @IO @String
  (Some hole fits suppressed;
   use -fmax-valid-hole-fits=N or -fno-max-valid-hole-fits)
```

getEnv has the exact type we are looking for, and sounds very much like the function we want.
We finish our definition by writing:

```haskell
module Main where
import System.Environment

getAuthCredentials :: IO (String, String)
getAuthCredentials = do key <- getEnv "VASTTRAFIKKEY"
                        secret <- getEnv "VASTTRAFIKSECRET"
                        return (key, secret)
main :: IO ()
main = getAuthCredentials >>= print
```

and test it,

```
$ ../inplace/bin/ghc-stage2 skjerm.hs
[1 of 1] Compiling Main             ( skjerm.hs, skjerm.o )
Linking skjerm ...
$ VASTTRAFIKKEY="TestKey" VASTTRAFIKSECRET="TestSecret" ./skjerm
("TestKey","TestSecret")
```

and it works!

4.6.1.2 Fetching the Token

To fetch the token we need to use the network. We do not want to deal with any `https` hassle ourselves, so we use the `req` library and add `Network.HTTP.Req` to our imports.
Now, we want to create a request, which is done by using the `req` function. To use the valid hole fits effectively, we need to know the return type, so let us launch GHCi to explore. The responses from GHC can be quite long, so we cut off the end of the messages with `...` in the interest of brevity.

```
*Main> import Network.HTTP.Req
*Main Network.HTTP.Req> :t req
req
  :: (HttpBodyAllowed (AllowsBody method) (ProvidesBody body),
      MonadHttp m, HttpMethod method, HttpBody body,
      HttpResponse response) =>
     method
     -> Url scheme
     -> body
     -> Data.Proxy.Proxy response
     -> Option scheme
     -> m response
```

We see that the return type should be a monad which has a `MonadHttp` instance, and the response needs to be a `HttpResponse`. But what types have those instances? Let us ask GHCi:

```
*Main Network.HTTP.Req> :i MonadHttp
class Control.Monad.IO.Class.MonadIO m =>
      MonadHttp (m :: * -> *) where
...
instance MonadHttp Req -- Defined in ‘Network.HTTP.Req’
*Main Network.HTTP.Req> :i HttpResponse
class HttpResponse response where
...
instance HttpResponse BsResponse -- Defined in ‘Network.HTTP.Req’
*Main Network.HTTP.Req>
```

Since we want a `ByteString`, we can use `Req BsResponse` as our target type. We proceed with the definition by focusing on each of the holes in order.
For the first hole, we write:

```
tokenRequest :: Req BsResponse
tokenRequest = req _ undefined undefined undefined undefined
```

which, when compiled, returns:

```
Valid hole fits include
  CONNECT :: CONNECT
    (imported from ‘Network.HTTP.Req’ at Main.hs:8:1-23)
  PATCH :: PATCH (imported from ‘Network.HTTP.Req’ at Main.hs:8:1-23)
  POST :: POST (imported from ‘Network.HTTP.Req’ at Main.hs:8:1-23)
  PUT :: PUT (imported from ‘Network.HTTP.Req’ at Main.hs:8:1-23)
  TRACE :: TRACE (imported from ‘Network.HTTP.Req’ at Main.hs:8:1-23)
```

We want to do a post request, so we choose POST and continue by writing:

```
tokenRequest :: Req BsResponse
tokenRequest = req POST _ undefined undefined undefined
```

For this next hole we get:

```
Found hole: _ :: Url scheme0
Where: ‘scheme0’ is an ambiguous type variable
In the second argument of ‘req’, namely ‘_’
In the expression: req POST _ undefined undefined undefined
In an equation for ‘tokenRequest’:
    tokenRequest = req POST _ undefined undefined undefined
Relevant bindings include
  tokenRequest :: Req BsResponse (bound at Main.hs:17:1)
Valid refinement hole fits include
  http (_ :: Data.Text.Internal.Text)
    where http :: Data.Text.Internal.Text -> Url 'Http
  https (_ :: Data.Text.Internal.Text)
    where https :: Data.Text.Internal.Text -> Url 'Https
  ...
```

We are doing a request to a `https` service, so we choose `https`, and proceed to the next hole by writing:

```
tokenRequest :: Req BsResponse
tokenRequest = req POST (https "api.vasttrafik.se/token") _ undefined undefined
```

Here we get:

```
Found hole: _ :: body0
Where: ‘body0’ is an ambiguous type variable
In the third argument of ‘req’, namely ‘_’
In the expression:
  req POST (https "api.vasttrafik.se/token") _ undefined undefined
In an equation for ‘tokenRequest’:
    tokenRequest
      = req POST (https "api.vasttrafik.se/token") _ undefined undefined
Relevant bindings include
  tokenRequest :: Req BsResponse (bound at Main.hs:19:1)
Valid hole fits include NoReqBody :: NoReqBody
Valid refinement hole fits include
  ReqBodyBs (_ :: ByteString)
    where ReqBodyBs :: ByteString -> ReqBodyBs
  ...
```

We want to send a ByteString specified by Västtraffik, so we choose ReqBodyBs, and proceed to the next hole by writing:

```
tokenRequest :: Req BsResponse
tokenRequest = req POST (https "api.vasttrafik.se/token")
                        (ReqBodyBs "grant_type=client_credentials")
                        _ undefined
```

Which gives us:

```
Found hole: _ :: Data.Proxy.Proxy BsResponse
In the fourth argument of ‘req’, namely ‘_’
In the expression:
  req POST (https "api.vasttrafik.se/token")
    (ReqBodyBs "grant_type=client_credentials")
    _ undefined
In an equation for ‘tokenRequest’:
    tokenRequest
      = req POST (https "api.vasttrafik.se/token")
          (ReqBodyBs "grant_type=client_credentials")
          _ undefined
Relevant bindings include
  tokenRequest :: Req BsResponse (bound at Main.hs:19:1)
Valid hole fits include
  bsResponse :: Data.Proxy.Proxy BsResponse
  ...
```

and since we are expecting a bytestring response, we choose bsResponse. We proceed to the final hole by writing:

```
tokenRequest :: Req BsResponse
tokenRequest = req POST (https "api.vasttrafik.se/token")
                        (ReqBodyBs "grant_type=client_credentials")
                        bsResponse _
```

and we get:

```
Found hole: _ :: Option 'Https
In the fifth argument of ‘req’, namely ‘_’
In the expression:
  req POST (https "api.vasttrafik.se/token")
    (ReqBodyBs "grant_type=client_credentials")
    bsResponse _
In an equation for ‘tokenRequest’:
    tokenRequest
      = req POST (https "api.vasttrafik.se/token")
          (ReqBodyBs "grant_type=client_credentials")
          bsResponse _
Relevant bindings include
  tokenRequest :: Req BsResponse (bound at Main.hs:19:1)
Valid hole fits include
  mempty :: forall a. Monoid a => a
    with mempty @(Option 'Https)
Valid refinement hole fits include
  oAuth2Bearer (_ :: ByteString)
    where oAuth2Bearer :: ByteString -> Option 'Https
  oAuth2Token (_ :: ByteString)
    where oAuth2Token :: ByteString -> Option 'Https
  basicAuth (_ :: ByteString) (_ :: ByteString)
    where basicAuth :: ByteString -> ByteString -> Option 'Https
  ...
```

Here we are setting the options for the request. Since the only thing we need to set is the basic authentication used to pass our key and secret, we choose `basicAuth`. Since we can not do `IO` at that point, we will just add them as arguments and finally arrive at:

```
tokenRequest :: ByteString -> ByteString -> Req BsResponse
tokenRequest key secret = req POST (https "api.vasttrafik.se/token")
                                   (ReqBodyBs "grant_type=client_credentials")
                                   bsResponse (basicAuth key secret)
```

But how do we perform the request? Let us check:

```
performReq :: Req BsResponse -> IO ByteString
performReq  = _
```

We compile this and get:

```
Found hole: _ :: Req BsResponse -> IO BsResponse
In the expression: _
In the expression: _ r
In an equation for ‘performReq’: performReq r = _ r
Relevant bindings include
  r :: Req BsResponse (bound at Main.hs:23:12)
  performReq :: Req BsResponse -> IO BsResponse
    (bound at Main.hs:23:1)
Valid hole fits include
  performReq :: Req BsResponse -> IO BsResponse
Valid refinement hole fits include
  runReq (_ :: HttpConfig)
    where runReq :: forall (m :: * -> *) a.
                    Control.Monad.IO.Class.MonadIO m =>
                    HttpConfig -> Req a -> m a
    with runReq @IO @BsResponse
  ...
```

Great, so we need to use the `runReq` with some configuration. But what can we use for configuration? We can ask GHC by writing:

```
performReq :: Req BsResponse -> IO BsResponse
performReq r = runReq _ r
```

When we compile this, we get:

```
Found hole: _ :: HttpConfig
In the first argument of ‘runReq’, namely ‘_’
In the expression: runReq _ r
In an equation for ‘performReq’: performReq r = runReq _ r
Relevant bindings include
  r :: Req BsResponse (bound at Main.hs:21:12)
  performReq :: Req BsResponse -> IO BsResponse
    (bound at Main.hs:21:1)
Valid refinement hole fits include
  id (_ :: HttpConfig)
    where id :: forall a. a -> a
    with id @HttpConfig
  Prelude.head (_ :: [HttpConfig])
    where Prelude.head :: forall a. [a] -> a
    with Prelude.head @HttpConfig
  Prelude.last (_ :: [HttpConfig])
    where Prelude.last :: forall a. [a] -> a
    with Prelude.last @HttpConfig
  asTypeOf (_ :: HttpConfig) (_ :: HttpConfig)
    where asTypeOf :: forall a. a -> a -> a
    with asTypeOf @HttpConfig
  (!!) (_ :: [HttpConfig]) (_ :: Int)
    where (!!) :: forall a. [a] -> Int -> a
    with (!!) @HttpConfig
```

This time, the valid hole fits do not help, and nothing useful is suggested. So we go into the documentation of the `req` package, and find that `HttpConfig` defines a `Data.Default` instance. It is not suggested, since `Data.Default` is not in the `Prelude`, and thus not in scope at the time of checking. If we add `Data.Default` to the imports, we get:

```
Found hole: _ :: HttpConfig
In the first argument of ‘runReq’, namely ‘_’
In the expression: runReq _ r
In an equation for ‘performReq’: performReq r = runReq _ r
Relevant bindings include
  r :: Req BsResponse (bound at Main.hs:22:12)
  performReq :: Req BsResponse -> IO BsResponse
    (bound at Main.hs:22:1)
Valid hole fits include
  def :: forall a. Default a => a
    with def @HttpConfig
```

and so we can fill the hole with `def`.

We are almost there, but first: How do we get a `ByteString` from a `BsResponse`? Let us ask GHC by compiling:

```
toBS :: BsResponse -> ByteString
toBS = _
```

GHC responds with the following message:

```
Found hole: _ :: BsResponse -> ByteString
In the expression: _
In an equation for ‘toBS’: toBS = _
Relevant bindings include
  toBS :: BsResponse -> ByteString (bound at Main.hs:25:1)
Valid hole fits include
  toBS :: BsResponse -> ByteString
  responseBody :: forall response.
                  HttpResponse response =>
                  response -> HttpResponseBody response
    with responseBody @BsResponse
  ...
```

which tells us that we can use `responseBody`.

Let us put it all together:

```
{-# LANGUAGE OverloadedStrings #-}
module Main where

import System.Environment

import Network.HTTP.Req
import Data.ByteString.Char8
import Data.Default

getAuthCredentials :: IO (String, String)
getAuthCredentials = do key <- getEnv "VASTTRAFIKKEY"
                        secret <- getEnv "VASTTRAFIKSECRET"
                        return (key, secret)

tokenRequest :: ByteString -> ByteString -> Req BsResponse
tokenRequest key secret = req POST (https "api.vasttrafik.se/token")
                               (ReqBodyBs "grant_type=client_credentials")
                               bsResponse (basicAuth key secret)

performReq :: Req BsResponse -> IO BsResponse
performReq r = runReq def r

toBS :: BsResponse -> ByteString
toBS = responseBody

main :: IO ()
main = do (key, secret) <- getAuthCredentials
          response <- performReq (tokenRequest (pack key) (pack secret))
          print $ toBS response
```

We now build and run with:

```
$ VASTTRAFIKKEY="REDACTED" VASTTRAFIKSECRET="REDACTED" ./PiSkjerm
```

But alas, it fails! The exception is the following message:

```
PiSkjerm: VanillaHttpException (HttpExceptionRequest Request {
  host                 = "api.vasttrafik.se%2Ftoken"
  port                 = 443
  secure               = True
  requestHeaders       = [("Authorization","<REDACTED>")]
  path                 = ""
  queryString          = ""
  method               = "POST"
  proxy                = Nothing
  rawBody              = False
  redirectCount        = 10
  responseTimeout      = ResponseTimeoutDefault
  requestVersion       = HTTP/1.1
}
```

It seems like the `url` is being encoded, so we need to find some other way to combine the domain and path part of the URL. Let us ask GHC:

```
tokenRequest :: ByteString -> ByteString -> Req BsResponse
tokenRequest key secret =
tokenRequest key secret = req POST (_ (https "api.vasttrafik.se") "token")
                                   (ReqBodyBs "grant_type=client_credentials")
                                   bsResponse (basicAuth key secret)
```

GHC tells us:

```
Found hole: _ :: Url 'Https -> [Char] -> Url 'Https
In the expression: _
In the second argument of ‘req’, namely
  ‘(_ (https "api.vasttrafik.se") "token")’
In the expression:
  req
    POST
    (_ (https "api.vasttrafik.se") "token")
    (ReqBodyBs "grant_type=client_credentials")
    bsResponse
    (basicAuth key secret)
Relevant bindings include
  secret :: ByteString (bound at Main.hs:18:18)
  key :: ByteString (bound at Main.hs:18:14)
  tokenRequest :: ByteString -> ByteString -> Req BsResponse
    (bound at Main.hs:18:1)
Valid hole fits include
  (/~) :: forall a (scheme :: Scheme).
          http-api-data-0.3.8.1:Web.Internal.HttpApiData.ToHttpApiData a =>
          Url scheme -> a -> Url scheme
    with (/~) @[Char] @'Https
  ...
```

so we fix the url part of token request to read:

```
((/~) @[Char] (https "api.vasttrafik.se") "token")
```

with the type application `@[Char]` to avoid issues with `OverloadedStrings`.
We now compile and run, and we get:

```
$ VASTTRAFIKKEY="REDACTED" VASTTRAFIKSECRET="REDACTED" ./PiSkjerm
"{\"scope\":\"am_application_scope default\",\"token_type\":\"Bearer\",
  \"expires_in\":3600,\"access_token\":\"REDACTED\"}"
```

It works! We are done.

4.6.2 Discussion

As we saw, the valid hole fits can get us quite far when working with real world libraries such as the `req` library. However, the onus of figuring out what modules to import is still on the programmer (such as when we import the `System.Environment` module when we get the credentials). We had to resort to GHCi to find which instances we were going to target, and it did not find the `def` due to `Data.Default` not being in scope, but otherwise we only had to let the types and the names of the functions guide us. The bug we encountered arose when the types were not specific enough, but the valid hole fits could help us out of that situation as well. My conclusion is that the valid hole fits can certainly improve the user experience when writing real world Haskell.

5 Related Work

5.1 Typed-Hole Suggestions in PureScript

PureScript is a statically typed functional programming language that compiles into JavaScript [31]. PureScript implements a type-directed search for suggestions when it encounters typed-holes, which was the inspiration for the valid hole fits in this thesis [32]. Similar to the feature in GHC, it searches through any imported libraries and the local module and uses subsumption to find values that work [32].
As an example the following PureScript snippet has a hole in `f` denoted with `?hole`:

```
module Main where

import Prelude

import Data.Monoid

lines :: String -> Array String
lines = mempty

words :: String -> Array String
words = mempty

f :: Array String
f = ?hole "hello, world"
```

When this snippet is compiled, the compiler outputs:

```
Hole 'hole' has the inferred type

    String -> Array String

  You could substitute the hole with one of these values:

    Control.Applicative.pure                        :: forall a f. Applicative f => a -> f a
    Control.Monad.Eff.Exception.Unsafe.unsafeThrow  :: forall a. String -> a
    Data.Array.singleton                            :: forall a. a -> Array a
    Data.Monoid.mempty                              :: forall m. Monoid m => m
    Data.Unfoldable.singleton                       :: forall f a. Unfoldable f => a -> f a
    Main.lines                                      :: String -> Array String
    Main.words                                      :: String -> Array String
    Partial.Unsafe.unsafeCrashWith                  :: forall a. String -> a
    Unsafe.Coerce.unsafeCoerce                      :: forall a b. a -> b


in value declaration f
```

In contrast to my implementation, the PureScript implementation does not implement refinement suggestions, though considering functions with additional arguments is mentioned in the future work section in Hegemann's thesis [32]. In addition, the list of suggestions is sorted by name, which puts very general suggestions like `unsafeThrow` earlier than exact matches such as `lines`, even though `lines` has the exact same type and is defined in the same module. This is also mentioned as a problem in the future work section of Hegemann's thesis [32].

5.2 Hoogle

Hoogle is a Haskell API search engine, which allows you to search many standard Haskell libraries by either function name, or by approximate type signature [33]. Hoogle is not based on using the GHC type checker, but uses information extracted from Haddock documentation stored in a binary database [33].
It has a few advantages over using the valid hole fits to search for valid fits:

- Hoogle searches for approximate types, and allows for the re-ordering of arguments and dropping of arguments, which the valid hole fits do not.
- Since the search does not rely on the GHC type checker, it is much faster.
- With its speed advantages, it can search all of the packages on Hackage for a match in reasonable time rather than just the functions that are in scope.
- Hoogle allows users to search by function name, which is useful if you want to achieve something specific, but are unsure what type it has.

However, as we saw in the evaluation, there are quite a few disadvantages as well:

- Since Hoogle is not based on the type checker, it can miss some very important details which are easy to handle when using the type checker, such as type synonyms.
- The search in Hoogle is not so good when it comes to expanded types, and cannot handle ambiguous type variables, and thus fails to find the `lens` operators as we saw in the evaluation.
- Hoogle is based on the Haddock documentation only, meaning that functions that are local to the module and un-exported functions are not included in the search [33].
- Hoogle does not run any type-family based computation, which means that use-cases such as the Sorting library are not possible with Hoogle.

Hoogle is a great tool, and will undoubtedly continue to be used in tandem with the valid hole fits.

5.3 Type-Driven Development in Dependently Typed Languages

As discussed in the introduction, typed-holes are more useful in dependently typed languages, since dependently typed languages can encode much more precise invariants on the type level than is possible in Haskell. However, being dependently typed means that type inference is undecidable, which means that more type signatures must be written by the programmer [10].

5.3.1 Typed-Holes in Agda

Agda is a dependently typed functional programming language based on intuitionistic type theory, which offers both typed-holes and automatic searching for fits for typed-holes [10, 34, 35]. The typed-holes of Agda inspired the typed-holes in GHC [4].

5.3.1.1 Example

Let us take a look at an example, in which we use the typed-holes in Agda [34]. We use the emacs `agda-mode`, which is the recommended way to program in Agda.
Imagine we have a data type for the natural numbers, and another data type which encodes a judgment of whether a natural number is even or not, that is a constructor that can only construct even numbers.

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ -> ℕ

data _even : ℕ -> Set where
   ZERO : zero even
   STEP : \forall x -> x even -> suc (suc x) even
```

Imagine that we want to prove that four is even:

```
proof : suc (suc (suc (suc zero))) even
proof = ?
```

When we invoke the "Load" command in `agda-mode` with `C-c C-l` to type check the program, Agda will convert the question mark into a hole, like this:

```
proof : suc (suc (suc (suc zero))) even
proof = { }0
```

And the following will appear in a separate Emacs buffer:

```
?0 : suc (suc (suc (suc zero))) even
```

This states that the type of hole `0` is `suc (suc (suc (suc zero))) even`.
Next, we can use the "Split obligation" command. The "Split obligation" command is invoked by placing the cursor into the hole and writing `STEP ? ?`, and then pressing `C-c C-space`. This splits the hole into two more holes:

```
proof : suc (suc (suc (suc zero))) even
proof = STEP { }1 { }2
```

```
?1 : ℕ
?2 : suc (suc zero) even
```

This tells us that for the first hole we have to provide something of the type of the natural numbers, and for the second hole we need to provide something of type `suc (suc zero) even`. If we put the cursor onto the second hole and run `C-C C-r` (the “refine” command), Agda will automatically figure out that there is only one constructor that fits that type, namely `STEP`. The `agda-mode` command will then replace the hole with that constructor and two additional holes, resulting in:

```
proof : suc (suc (suc (suc zero))) even
proof = STEP { }1 (STEP { }3 { }4)
```

```
?1 : ℕ
?3 : ℕ
?4 : zero even
```

Placing the cursor into hole number 4 and running `C-c C-r` will replace it with the only constructor of type `zero even`, namely `ZERO`.
Resulting in:

```
proof : suc (suc (suc (suc zero))) even
proof = STEP { }1 (STEP { }3 ZERO)
```

```
?1 : ℕ
?3 : ℕ
```

The remaining holes can be solved automatically from the constraints on them. These constraints can be shown by running the “Show constraints” command, invoked by typing `C-c C-=`. This will in our case print out:

```
?1 : suc (suc zero)
?3 : zero
```

If we invoke the “Solve constraints” command with `C-c C-s`, Agda will now fill these holes with the only valid options, namely 2 and 0 [34].

5.3.1.2 The Auto command in Agda

Agda can also do a search for valid hole fits, with the Auto command. Formerly known as Agsy, it can be invoked in `agda-mode` by hovering over a hole and invoking it with `C-c C-a` [34, 35]. It will then search for type inhabitants and fill the hole if found [35].
As in the example above, if we have:

```
proof : suc (suc (suc (suc zero))) even
proof = STEP { }1 (STEP zero ZERO)
```

Then placing the cursor inside `{ }1` and invoking the auto command with `C-c C-a` would replace the hole with `(suc (suc zero))` resulting in:

```
proof : suc (suc (suc (suc zero))) even
proof = STEP (suc (suc zero)) (STEP zero ZERO)
```

The Auto command is quite powerful, and works for small enough problems of almost any kind. It can construct terms beyond single identifiers or identifiers applied to additional holes. It can attempt case-splitting and refinement, and it can even do equality reasoning [35]. By default, it replaces the hole with the first solution found, but it can also list valid solutions. It can take "hints", which can be identifiers or definitions in the current module that can help the Auto command find a match quicker [35].

5.3.1.3 Discussion

As discussed in the introduction, a dependently typed setting brings a lot more power to typed-holes beyond what is possible in Haskell. However, this comes at the price of much more explicit typing of functions and values by the programmer.

5.3.2 Idris

Type-Driven development is an approach that is highly encouraged when working with Idris, and indeed the title of the primary text on Idris is Type-Driven Development with Idris, which is written by Edwin Brady the author of Idris [3].
Idris also has typed-holes, which can be placed and inspected in the Idris REPL [3]. As an example, one could write:

```
module Main

main : IO ()
main = putStrLn (?convert 'x')
```

When loaded into the REPL, the user can inspect the hole:

```
*Hello> :t convert
--------------------------------
convert : Char -> String
```

Holes can be automatically lifted to top-level functions in Atom, the editor of choice for Idris, with a built-in command [3].
Idris also implements expression search, which allows users to search for expressions with a given type either from the command line or by invoking `Ctrl-Alt-S` on a hole [3]. The `search` command is based on type isomorphism, and as such can find types such as `b -> a -> a` when searching for `a -> b -> a` [36].

```
Idris> :search e -> List e -> List e
= Prelude.List.(::) : a -> List a -> List a
Cons cell

= Prelude.List.intersperse : a -> List a -> List a
Insert some separator between the elements of a list.

> Prelude.List.delete : Eq a => a -> List a -> List a


< assert_smaller : a -> b -> b
Assert to the totality checker than y is always structurally
smaller than x (which is typically a pattern argument)

< Prelude.Basics.const : a -> b -> a
Constant function. Ignores its second argument.
```

The list displays functions that are either exact matches (denoted with `=`), more specific matches (denoted with <) and more general matches (denoted with >) [36]. The ordering is based on the "edit distance", i.e. the length of the chain of rewrite rules needed to get from one type to the other [36].

Idris also has the `:proofsearch` command, which works similarly to the `Auto` command in Agda, and constructs terms of the correct type [36].

5.3.2.1 Discussion

Idris has very powerful tools for type-driven development, and there are certainly ideas there that could be used to improve the typed-holes in GHC even further. Showing more specific types is something that would definitely be useful in GHC, and would resolve our problem with GHC inferring types which are too general as discussed in the first item of future work in section 6.2, and encountered in our evaluation of the free monad example in section 4.4. This could be implemented in our extension by running an inverted subsumption check in addition to the regular subsumption check when looking for valid hole fits. We explore this idea further in the second item of the future work section, 6.2.

5.3.3 Liquid Haskell

Liquid Haskell does not have any typed-hole features at the term level beyond those already present in GHC, and does not offer any `proofsearch` or `type-directed search` like `Agda` and `Idris` do. Liquid Haskell does however have holes at the type level, similar to the `PartialTypeSignatures` extensions. This allows users to use underscores to represent holes in refinement types, which Liquid Haskell can then replace with types inferred from GHC or gleaned from the type signature if possible [11].

5.4 Program Synthesis

Finding valid hole fits for typed-holes can be viewed as simplistic program synthesis, that considers only one term (which might contain additional holes) and only takes into account a target type. A number of type-directed program synthesizers exists, many of which target completion of partial expressions like valid hole fits. These include:

- Prospector is an early type-directed program synthesizer that targets completion, and predates the other synthesizers presented here. Prospector helps users navigate the API jungle by synthesizing *jungloids* (unary expressions) in response to user queries from an input type to an output type [37]. These jungloids are synthesized based on a type graph built from API method signatures, *mined* jungloids from a corpus of programs and downcasts, and are ranked by the distance between the input and output type within this graph [37]. Prospector was originally integrated with the Eclipse IDE, but does not seem to be publicly available any more [37].
- Perelman et al. developed an algorithm for type-directed completion of partial expressions for C#/Java [38]. Their algorithm allows for information to be omitted, similarly to the additional holes in refinement hole fits [38]. They distinguish between guessing method names and method arguments, while in our case both are simply identifiers with different types [38]. They found that type distance was the most important metric when ranking completions [38].
- InSynth offers type-directed completion for Java and Scala, based only on the type [39]. InSynth generates entire expressions that can include higher order functions [39]. InSynth uses a machine learning based approach that uses weights to rank completions, and these weights can be assigned by the user or mined from a corpus of code [39]. InSynth has been deployed in an IDE for Scala [39].
- Djinn is a command line tool (not integrated with GHC) that generates Haskell code from a type [40]. In Djinn, users have to manually add types and functions to be considered, and it does not instantiate polymorphic functions [40]. Djinn is available from Hackage [40].
- Synquid is an algorithm developed by Polikarpova et al., that can synthesize programs from polymorphic refinement types in an ML-like language [41]. By leveraging the more expressive refinement types, they can synthesize programs that provably satisfy a rich specification, such as sorting algorithms and BST operations [41]. Synquid is available as a command line tool and web interface from the Synquid repository [41].

None of these is integrated into a language or with the type checker of a language, but are rather stand-alone tools or IDE plugins.

6 Conclusion

As we can see from the evaluation, the valid hole fits extension improves the user experience and facilitates a type-driven development approach. It does this in a lightweight way for a variety of practical scenarios for both newcomers and advanced users, such as when working with real world libraries such as `lens` or `req` or more advanced categorical objects such as the free monad. The implementation makes use of the already present type checking mechanisms of GHC, and integrates well with the typed-holes in a non-intrusive manner.

6.1 Impact

My contributions to the Glasgow Haskell Compiler have been accepted, with some available from version 8.4.1, while later additions such as local bindings, refinement hole fits and sorting of the output will be available starting with version 8.6.1.

I had a poster on the topic accepted into the ACM student research competition at ICFP 2017. While there I got insightful feedback on how to extend the feature to handle more general cases than simple type equality, and worked with Simon Peyton Jones on the original implementation. I also gave a talk at Lambda Days 2018, where I explained typed-holes to a non-expert audience and showed how the valid hole fit suggestions could be used in practice. I received great feedback on the feature from both beginners and advanced users alike, which informed the further development of the feature.

6.2 Future Work

- One issue that comes to light when working with typed-holes as seen in the evaluation is that the type inferred by GHC might be too polymorphic for the valid hole fits to be useful, as we encountered in the evaluation in section 4.4. As an example, consider the function `f x = (_+x)/5`. Here, GHC will infer that `f :: Fractional a => a -> a`. However, `pi :: Floating a => a` is a valid fit for the hole in `f`, but only if the inferred type of `f` was the stricter `f :: Floating a => a`. Since we use the type that GHC infers in this case, it is not suggested as a valid hole fit. If this were the first time we are referencing `f`, giving it a more strict type would not be a problem, but we might have written

```haskell
a :: Floating a => a
a = f 1

b :: Fractional a => a
b = f 2
```

and then filling in the hole with `pi` would make the program invalid! What is worse is that these invocations are not limited to even the module, but could have an effect in another module or even another package!
One option is to suggest these fits only in the case that we are looking for a fit for a hole for a local binding and not a global element, or suggesting that the user enable `-XMonoLocalBinds` like we did in the evaluation of the free monad example. Then we could be sure that the suggestion would not change the meaning outside the scope we are considering.
Another is to suggest to the programmer that they *could* use this fit if the constraints were tightened to the new constraints, but warning that this may not be valid.

- As mentioned in the discussion on the type-directed search feature of Idris, we could additionally show more specific fits than the type of the hole by also running an inverted subsumption check in addition to the regular subsumption check. Then, if the inverted subsumption check is a match, this means that the type of the match is more specific than the type of the hole. These matches could then be output as a separate list of suggestions. These fits would not be valid in all cases (as discussed in the item above), but they could certainly be useful in many cases.
- As we encountered during our evaluation of exercises from Hutton in section 4.3, the current implementation does not consider `(:)` and other functions that are treated as built-in syntax. However, these can often be the identifier that the user is looking for, and would improve the user experience if added.
- IDE integration. The valid hole fits would be more useful to the programmer if it were integrated into their IDE. One such approach would be enhanced auto–completion based not just on names (as they usually are), but also on the type itself. For this to work, the compilation process would need to be very quick, and the IDE able to ask for the output for this particular expression quickly.
- Adding valid hole fits to Liquid Haskell. One problem with the valid hole fits is that they only care about the type of the hole, with no way to hint to the compiler that the desired function must have some specified behavior. One solution would be to integrate the valid hole fits into the Liquid Haskell compiler. In Liquid Haskell, refinement types are used to provide stricter guarantees than the current type system of Haskell can offer by encoding invariants on the type level [42]. In Liquid Haskell, these are expressed via additional annotations to functions, for example

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

  which states that the `isPositive` function only returns true if its input `x` is positive [42]. By integrating similar checks into the valid hole fits, we could suggest only those functions which fit these more precise refinement types, similar to how SYNQUID operates (but within Haskell) [41].

6.3 Reflection

I certainly learned a lot from this project. I gained a lot of experience in working with and extending GHC, as well as a good understanding of some parts of the type checker. I now know how to effectively work with the latest GHC from the HEAD branch, and how to get it to work even when building libraries with a lot of dependencies such as `lens`. During the PiSkjerm project, I also became acquainted with cross-compilation of Haskell programs for the Raspberry Pi, contributed to the mobile Haskell user guide, and contributed a Docker image to make cross-compilation easier for others. I am also proud of having contributed to GHC and added a feature that has been well received by the community.

Bibliography

[1] B. C. Pierce, *Types and programming languages.* MIT press, 2002.

[2] GitHub, "The Open Source Survey," 2017. [Online]. Available: opensourcesurvey.org/2017/

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

[4] Haskell Wiki Contributors, "Typed holes in GHC," 2014. [Online]. Available: wiki.haskell.org/index.php?title=GHC/Typed_holes&oldid=58717

[5] Stack Overflow, "Developer survey," 2017. [Online]. Available: insights.stackoverflow.com/survey/2017/

[6] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the Scala programming language," EPFL Lausanne, Tech. Rep., 2004.

[7] S. P. Jones, D. Vytiniotis, S. Weirich, and M. Shields, "Practical type inference for arbitrary-rank types," *Journal of Functional Programming*, vol. 17, no. 1, pp. 1–82, 2007.

[8] Haskell Weekly, "State of Haskell survey," 2017. [Online]. Available: taylor.fausak.me/2017/11/15/2017-state-of-haskell-survey-results/

[9] A. Bove and P. Dybjer, "Dependent types at work," in *Language engineering and rigorous software development.* Springer, 2009, pp. 57–99.

[10] U. Norell, "Dependently typed programming in Agda," in *International School on Advanced Functional Programming.* Springer, 2008, pp. 230–266.

[11] N. Vazou, E. L. Seidel, and R. Jhala, "LiquidHaskell: Experience with Refinement types in the Real World," in *Haskell Symposium.* ACM, 2014.

[12] R. Eisenberg, "Dependent Types in Haskell: Theory and Practice," Ph.D. dissertation, University of Pennsylvania, 2016.

[13] M. Sulzmann, "A general type inference framework for Hindley/Milner style systems," in *International Symposium on Functional and Logic Programming.* Springer, 2001, pp. 248–263.

[14] S. Marlow *et al.*, "Haskell 2010 language report," 2010. [Online]. Available: www.haskell.org/definition/haskell2010.pdf

[15] M. Sulzmann, M. M. Chakravarty, S. P. Jones, and K. Donnelly, "System F with type equality coercions," in *International Workshop on Types in Languages Design and Implementation.* ACM, 2007, pp. 53–66.

[16] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *POPL '89, Proc. the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages.* ACM, 1989, pp. 60–76.

[17] G. Hutton, *Programming in Haskell.* Cambridge University Press, 2016.

[18] GHC Contributors, "GHC 8.2.1 users guide," 2017. [Online]. Available: downloads.haskell.org/~ghc/8.2.1/docs/html/users_guide/index.html

[19] E. Kmett, "The lens library," 2018. [Online]. Available: hackage.haskell.org/package/lens

[20] S. Marlow and S. P. Jones, "The Glasgow Haskell Compiler," 2012. [Online]. Available: microsoft.com/en-us/research/wp-content/uploads/2012/01/aos.pdf

[21] A. Ranta, *Implementing programming languages. An introduction to compilers and interpreters.* College Publications, 2012.

[22] GHC Contributors, "GHC 8.4.2 source code," 2018. [Online]. Available: github.com/ghc/ghc/tree/28595b7ab619d9a812cce23a546d7deabb486372

[23] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis, "Complete and decidable type inference for GADTs," in *ICFP '09, Proc. the 14th ACM SIGPLAN International Conference on Functional Programming.* ACM, 2009, pp. 341–352.

[24] G. S. Boolos, J. P. Burgess, and R. C. Jeffrey, *Computability and Logic.* Cambridge university press, 2007.

[25] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann, "Type checking with open type functions," in *ICFP '08, Proc. the 13th ACM SIGPLAN International Conference on Functional Programming.* ACM, 2008, pp. 51–62.

[26] N. Vazou, "Liquid Haskell: Haskell as a theorem prover," Ph.D. dissertation, University of California, San Diego, 2016.

[27] V. Trnková, J. Adámek, V. Koubek, and J. Reiterman, "Free algebras, input processes and free monads," *Commentationes Mathematicae Universitatis Carolinae*, vol. 16, no. 2, pp. 339–351, 1975.

[28] S. Mac Lane, *Categories for the working mathematician.* Springer Science & Business Media, 2013, vol. 5.

[29] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in Haskell," in *ICFP '98, Proc. the 3rd ACM SIGPLAN International Conference on Functional Programming.* ACM, 1998, pp. 174–184.

[30] M. Algehed and A. Russo, "Encoding DCC in Haskell," in *PLAS '17, Proc. 2017 Workshop on Programming Languages and Analysis for Security.* ACM, 2017, pp. 77–89.

[31] P. Freeman, *PureScript by Example.* Leanpub, 2017.

[32] C. Hegemann, "Implementing type directed search for PureScript," 2016, BSc. Thesis, University of Applied Sciences, Cologne.

[33] N. Mitchell, "Hoogle overview," *The Monad. Reader*, vol. 12, pp. 27–35, 2008.

[34] W. DeMeo, "Learn you an Agda," 2014. [Online]. Available: williamdemeo.github.io/2014/02/27/learn-you-an-agda/

[35] Agda Contributors, "Automatic Proof Search (Auto) Documentation 2.5.3," 2017. [Online]. Available: agda.readthedocs.io/en/v2.5.3/tools/auto.html

[36] The Idris Community, "Idris Documentation," 2017. [Online]. Available: docs.idris-lang.org

[37] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *PLDI '05, Proc. the 26th ACM SIGPLAN Con-*

ference on Programming Language Design and Implementation. ACM, 2005, pp. 48–61.

[38] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in *PLDI '12, Proc. the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 2012, pp. 275–286.

[39] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete completion using types and weights," in *PLDI '13, Proc. the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 2013, pp. 27–38.

[40] L. Augustsson, "The Djinn package," 2014. [Online]. Available: hackage.haskell.org/package/lens

[41] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, "Program synthesis from polymorphic refinement types," in *PLDI '16, Proc. the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 2016, pp. 522–538.

[42] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones, "Refinement types for Haskell," in *ICFP '14, Proc. the 19th ACM SIGPLAN International Conference on Functional Programming.* ACM, 2014, pp. 269–282.

A Appendix

A.1 The Implementation

My contributions to GHC were originally spread around several modules, but they were consolidated to one module as the extension grew in scope. The `TcHoleErrors` defines the `findValidHoleFits` functions which the hole error reporter calls in `TcErrors`.

```
{-# OPTIONS_GHC -fno-warn-orphans #-} -- We don't want to spread the HasOccName
                                      -- instance for Either
module TcHoleErrors ( findValidHoleFits ) where

import GhcPrelude

import TcRnTypes
import TcRnMonad
import TcMType
import TcEvidence
import TcType
import Type
import DataCon
import Name
import RdrName ( pprNameProvenance , GlobalRdrElt (..), globalRdrEnvElts )
import PrelNames ( gHC_ERR )
import Id
import VarSet
import VarEnv
import Bag
import ConLike          ( ConLike(..) )
import Util
import TcEnv (tcLookup)
import Outputable
import DynFlags
import Maybes
import FV ( fvVarList, fvVarSet, unionFV, mkFVs, FV )

import Control.Arrow ( (&&&) )

import Control.Monad    ( filterM, replicateM )
import Data.List        ( partition, sort, sortOn, nubBy, foldl' )
import Data.Graph       ( graphFromEdges, topSort )
import Data.Function    ( on )


import TcSimplify    ( simpl_top, runTcSDeriveds )
import TcUnify       ( tcSubType_NC )

{-
Note [Valid hole fits include ...]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
‘findValidHoleFits‘ returns the "Valid hole fits include ..." message.
For example, look at the following definitions in a file called test.hs:

   import Data.List (inits)

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

The hole in ‘f‘ would generate the message:

  • 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 test.hs:6:1)
    Valid hole fits include
      lines :: String -> [String]
        (imported from ‘Prelude’ at mpt.hs:3:8-9
          (and originally defined in ‘base-4.11.0.0:Data.OldList’))
      words :: String -> [String]
        (imported from ‘Prelude’ at mpt.hs:3:8-9
          (and originally defined in ‘base-4.11.0.0:Data.OldList’))
      inits :: forall a. [a] -> [[a]]
        with inits @Char
        (imported from ‘Data.List’ at mpt.hs:4:19-23
          (and originally defined in ‘base-4.11.0.0:Data.OldList’))
      repeat :: forall a. a -> [a]
        with repeat @String
        (imported from ‘Prelude’ at mpt.hs:3:8-9
          (and originally defined in ‘GHC.List’))
      fail :: forall (m :: * -> *). Monad m => forall a. String -> m a
        with fail @[] @String
        (imported from ‘Prelude’ at mpt.hs:3:8-9
          (and originally defined in ‘GHC.Base’))
      return :: forall (m :: * -> *). Monad m => forall a. a -> m a
        with return @[] @String
        (imported from ‘Prelude’ at mpt.hs:3:8-9
          (and originally defined in ‘GHC.Base’))
      pure :: forall (f :: * -> *). Applicative f => forall a. a -> f a
        with pure @[] @String
        (imported from ‘Prelude’ at mpt.hs:3:8-9
          (and originally defined in ‘GHC.Base’))
      read :: forall a. Read a => String -> a
        with read @[String]
        (imported from ‘Prelude’ at mpt.hs:3:8-9
          (and originally defined in ‘Text.Read’))
      mempty :: forall a. Monoid a => a
        with mempty @([Char] -> [String])
        (imported from ‘Prelude’ at mpt.hs:3:8-9
          (and originally defined in ‘GHC.Base’))
```

```
Valid hole fits are found by checking top level identifiers and local bindings
in scope for whether their type can be instantiated to the the type of the hole.
Additionally, we also need to check whether all relevant constraints are solved
by choosing an identifier of that type as well, see Note [Relevant Constraints]

Since checking for subsumption results in the side-effect of type variables
being unified by the simplifier, we need to take care to restore them after
to being flexible type variables after we've checked for subsumption.
This is to avoid affecting the hole and later checks by prematurely having
unified one of the free unification variables.

When outputting, we sort the hole fits by the size of the types we'd need to
apply by type application to the type of the fit to to make it fit. This is done
in order to display "more relevant" suggestions first. Another option is to
sort by building a subsumption graph of fits, i.e. a graph of which fits subsume
what other fits, and then outputting those fits which are are subsumed by other
fits (i.e. those more specific than other fits) first. This results in the ones
"closest" to the type of the hole to be displayed first.

To help users understand how the suggested fit works, we also display the values
that the quantified type variables would take if that fit is used, like
`mempty @([Char] -> [String])` and `pure @[] @String` in the example above.
If -XTypeApplications is enabled, this can even be copied verbatim as a
replacement for the hole.


Note [Nested implications]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

For the simplifier to be able to use any givens present in the enclosing
implications to solve relevant constraints, we nest the wanted subsumption
constraints and relevant constraints within the enclosing implications.

As an example, let's look at the following code:

  f :: Show a => a -> String
  f x = show _

The hole will result in the hole constraint:

  [WD] __a1ph {0}:: a0_a1pd[tau:2] (CHoleCan: ExprHole(_))

Here the nested implications are just one level deep, namely:

  [Implic {
      TcLevel = 2
      Skolems = a_a1pa[sk:2]
      No-eqs = True
      Status = Unsolved
      Given = $dShow_a1pc :: Show a_a1pa[sk:2]
      Wanted =
        WC {wc_simple =
              [WD] __a1ph {0}:: a_a1pd[tau:2] (CHoleCan: ExprHole(_))
              [WD] $dShow_a1pe {0}:: Show a_a1pd[tau:2] (CDictCan(psc))}
      Binds = EvBindsVar<a1pi>
      Needed inner = []
```

```
      Needed outer = []
      the type signature for:
        f :: forall a. Show a => a -> String }]

As we can see, the givens say that the information about the skolem
‘a_a1pa[sk:2]‘ fulfills the Show constraint.

The simples are:

  [[WD] __a1ph {0}:: a0_a1pd[tau:2] (CHoleCan: ExprHole(_)),
    [WD] $dShow_a1pe {0}:: Show a0_a1pd[tau:2] (CNonCanonical)]

I.e. the hole ‘a0_a1pd[tau:2]‘ and the constraint that the type of the hole must
fulfill ‘Show a0_a1pd[tau:2])‘.

So when we run the check, we need to make sure that the

  [WD] $dShow_a1pe {0}:: Show a0_a1pd[tau:2] (CNonCanonical)

Constraint gets solved. When we now check for whether ‘x :: a0_a1pd[tau:2]‘ fits
the hole in ‘tcCheckHoleFit‘, the call to ‘tcSubType‘ will end up writing the
meta type variable ‘a0_a1pd[tau:2] := a_a1pa[sk:2]‘. By wrapping the wanted
constraints needed by tcSubType_NC and the relevant constraints (see
Note [Relevant Constraints] for more details) in the nested implications, we
can pass the information in the givens along to the simplifier. For our example,
we end up needing to check whether the following constraints are soluble.

  WC {wc_impl =
        Implic {
          TcLevel = 2
          Skolems = a_a1pa[sk:2]
          No-eqs = True
          Status = Unsolved
          Given = $dShow_a1pc :: Show a_a1pa[sk:2]
          Wanted =
            WC {wc_simple =
                  [WD] $dShow_a1pe {0}:: Show a0_a1pd[tau:2] (CNonCanonical)}
          Binds = EvBindsVar<a1pl>
          Needed inner = []
          Needed outer = []
          the type signature for:
            f :: forall a. Show a => a -> String }}

But since ‘a0_a1pd[tau:2] := a_a1pa[sk:2]‘ and we have from the nested
implications that Show a_a1pa[sk:2] is a given, this is trivial, and we end up
with a final WC of WC {}, confirming x :: a0_a1pd[tau:2] as a match.

To avoid side-effects on the nested implications, we create a new EvBindsVar so
that any changes to the ev binds during a check remains localised to that check.


Note [Valid refinement hole fits include ...]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
When the ‘-frefinement-level-hole-fits=N‘ flag is given, we additionally look
for "valid refinement hole fits"", i.e. valid hole fits with up to N
additional holes in them.
```

```
With ‘-frefinement-level-hole-fits=0‘ (the default), GHC will find all
identifiers ’f’ (top-level or nested) that will fit in the hole.

With ‘-frefinement-level-hole-fits=1‘, GHC will additionally find all
applications ’f _’ that will fit in the hole, where ’f’ is an in-scope
identifier, applied to single argument.  It will also report the type of the
needed argument (a new hole).

And similarly as the number of arguments increases

As an example, let’s look at the following code:

  f :: [Integer] -> Integer
  f = _

with ‘-frefinement-level-hole-fits=1‘, we’d get:

  Valid refinement hole fits include

    foldl1 (_ :: Integer -> Integer -> Integer)
      with foldl1 @[] @Integer
      where foldl1 :: forall (t :: * -> *).
                      Foldable t =>
                      forall a. (a -> a -> a) -> t a -> a
    foldr1 (_ :: Integer -> Integer -> Integer)
      with foldr1 @[] @Integer
      where foldr1 :: forall (t :: * -> *).
                      Foldable t =>
                      forall a. (a -> a -> a) -> t a -> a
    const (_ :: Integer)
      with const @Integer @[Integer]
      where const :: forall a b. a -> b -> a
    ($) (_ :: [Integer] -> Integer)
      with ($) @’GHC.Types.LiftedRep @[Integer] @Integer
      where ($) :: forall a b. (a -> b) -> a -> b
    fail (_ :: String)
      with fail @((->) [Integer]) @Integer
      where fail :: forall (m :: * -> *).
                    Monad m =>
                    forall a. String -> m a
    return (_ :: Integer)
      with return @((->) [Integer]) @Integer
      where return :: forall (m :: * -> *). Monad m => forall a. a -> m a
    (Some refinement hole fits suppressed;
      use -fmax-refinement-hole-fits=N or -fno-max-refinement-hole-fits)

Which are hole fits with holes in them. This allows e.g. beginners to
discover the fold functions and similar, but also allows for advanced users
to figure out the valid functions in the Free monad, e.g.

  instance Functor f => Monad (Free f) where
      Pure a >>= f = f a
      Free f >>= g = Free (fmap _a f)

Will output (with -frefinment-level-hole-fits=1):
    Found hole: _a :: Free f a -> Free f b
          Where: ‘a’, ‘b’ are rigid type variables bound by
                  the type signature for:
                    (>>=) :: forall a b. Free f a -> (a -> Free f b) -> Free f b
                  at fms.hs:25:12-14
                ‘f’ is a rigid type variable bound by
    ...
    Relevant bindings include
      g :: a -> Free f b (bound at fms.hs:27:16)
      f :: f (Free f a) (bound at fms.hs:27:10)
      (>>=) :: Free f a -> (a -> Free f b) -> Free f b
        (bound at fms.hs:25:12)
    ...
    Valid refinement hole fits include
      ...
      (=<<) (_ :: a -> Free f b)
        with (=<<) @(Free f) @a @b
        where (=<<) :: forall (m :: * -> *) a b.
                       Monad m =>
                       (a -> m b) -> m a -> m b
        (imported from ‘Prelude’ at fms.hs:5:18-22
        (and originally defined in ‘GHC.Base’))
      ...

Where ‘(=<<) _‘ is precisely the function we want (we ultimately want ‘>>= g‘).

We find these refinement suggestions by considering hole fits that don’t
fit the type of the hole, but ones that would fit if given an additional
argument. We do this by creating a new type variable with ‘newOpenFlexiTyVar‘
(e.g. ‘t_a1/m[tau:1]‘), and then considering hole fits of the type
‘t_a1/m[tau:1] -> v‘ where ‘v‘ is the type of the hole.

Since the simplifier is free to unify this new type variable with any type, we
can discover any identifiers that would fit if given another identifier of a
suitable type. This is then generalized so that we can consider any number of
additional arguments by setting the ‘-frefinement-level-hole-fits‘ flag to any
number, and then considering hole fits like e.g. ‘foldl _ _‘ with two additional
arguments.

To make sure that the refinement hole fits are useful, we check that the types
of the additional holes have a concrete value and not just an invented type
variable. This eliminates suggestions such as ‘head (_ :: [t0 -> a]) (_ :: t0)‘,
and limits the number of less than useful refinement hole fits.

Additionally, to further aid the user in their implementation, we show the
types of the holes the binding would have to be applied to in order to work.
In the free monad example above, this is demonstrated with
‘(=<<) (_ :: a -> Free f b)‘, which tells the user that the ‘(=<<)‘ needs to
be applied to an expression of type ‘a -> Free f b‘ in order to match.
If -XScopedTypeVariables is enabled, this hole fit can even be copied verbatim.


Note [Relevant Constraints]
~~~~~~~~~~~~~~~~~~~

As highlighted by Trac #14273, we need to check any relevant constraints as well
```

```
as checking for subsumption. Relevant constraints are the simple constraints
whose free unification variables are mentioned in the type of the hole.

In the simplest case, these are all non-hole constraints in the simples, such
as is the case in

  f :: String
  f = show _

Where the simples will be :

  [[WD] __a1kz {0}:: a0_a1kv[tau:1] (CHoleCan: ExprHole(_)),
    [WD] $dShow_a1kw {0}:: Show a0_a1kv[tau:1] (CNonCanonical)]

However, when there are multiple holes, we need to be more careful. As an
example, Let's take a look at the following code:

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

Here there are two holes, `_a` and `_b`, and the simple constraints passed to
findValidHoleFits are:

  [[WD] _a_a1pi {0}:: String
                         -> a0_a1pd[tau:2] (CHoleCan: ExprHole(_b)),
    [WD] _b_a1ps {0}:: a1_a1po[tau:2] (CHoleCan: ExprHole(_a)),
    [WD] $dShow_a1pe {0}:: Show a0_a1pd[tau:2] (CNonCanonical),
    [WD] $dShow_a1pp {0}:: Show a1_a1po[tau:2] (CNonCanonical)]


Here we have the two hole constraints for `_a` and `_b`, but also additional
constraints that these holes must fulfill. When we are looking for a match for
the hole `_a`, we filter the simple constraints to the "Relevant constraints",
by throwing out all hole constraints and any constraints which do not mention
a variable mentioned in the type of the hole. For hole `_a`, we will then
only require that the `$dShow_a1pp` constraint is solved, since that is
the only non-hole constraint that mentions any free type variables mentioned in
the hole constraint for `_a`, namely `a_a1pd[tau:2]` , and similarly for the
hole `_b` we only require that the `$dShow_a1pe` constraint is solved.

-}



data HoleFitDispConfig = HFDC { showWrap :: Bool
                              , showWrapVars :: Bool
                              , showType :: Bool
                              , showProv :: Bool
                              , showMatches :: Bool }

debugHoleFitDispConfig :: HoleFitDispConfig
debugHoleFitDispConfig = HFDC True True True False False


-- We read the various -no-show-*-of-hole-fits flags
-- and set the display config accordingly.
getHoleFitDispConfig :: TcM HoleFitDispConfig
getHoleFitDispConfig
  = do { sWrap <- goptM Opt_ShowTypeAppOfHoleFits
       ; sWrapVars <- goptM Opt_ShowTypeAppVarsOfHoleFits
       ; sType <- goptM Opt_ShowTypeOfHoleFits
       ; sProv <- goptM Opt_ShowProvOfHoleFits
       ; sMatc <- goptM Opt_ShowMatchesOfHoleFits
       ; return HFDC{ showWrap = sWrap, showWrapVars = sWrapVars
                    , showProv = sProv, showType = sType
                    , showMatches = sMatc } }

-- Which sorting algorithm to use
data SortingAlg = NoSorting      -- Do not sort the fits at all
                | BySize         -- Sort them by the size of the match
                | BySubsumption  -- Sort by full subsumption
                deriving (Eq, Ord)

getSortingAlg :: TcM SortingAlg
getSortingAlg =
    do { shouldSort <- goptM Opt_SortValidHoleFits
       ; subsumSort <- goptM Opt_SortBySubsumHoleFits
       ; sizeSort <- goptM Opt_SortBySizeHoleFits
       -- We default to sizeSort unless it has been explicitly turned off
       -- or subsumption sorting has been turned on.
       ; return $ if not shouldSort
                    then NoSorting
                    else if subsumSort
                         then BySubsumption
                         else if sizeSort
                              then BySize
                              else NoSorting }

-- HoleFit is the type we use for valid hole fits. It contains the
-- element that was checked, the Id of that element as found by ‘tcLookup‘,
-- and the refinement level of the fit, which is the number of extra argument
-- holes that this fit uses (e.g. if hfRefLvl is 2, the fit is for ‘Id _ _‘).
data HoleFit = HoleFit { hfElem :: Maybe GlobalRdrElt -- The element that was
                                                       -- if a global, nothing
                                                       -- if it is a local.
                       , hfId :: Id       -- The elements id in the TcM
                       , hfType :: TcType -- The type of the id, possibly zonked
                       , hfRefLvl :: Int  -- The number of holes in this fit
                       , hfWrap :: [TcType] -- The wrapper for the match
                       , hfMatches :: [TcType] } -- What the refinement
                                                 -- variables got matched with,
                                                 -- if anything

-- We define an Eq and Ord instance to be able to build a graph.
instance Eq HoleFit where
   (==) = (==) ‘on‘ hfId

-- We compare HoleFits by their gre_name instead of their Id, since we don’t
-- want our tests to be affected by the non-determinism of ‘nonDetCmpVar‘,
-- which is used to compare Ids. When comparing, we want HoleFits with a lower
-- refinement level to come first.
```

```haskell
instance Ord HoleFit where
  compare a b = cmp a b
    where cmp  = if hfRefLvl a == hfRefLvl b
                 then compare `on` (idName . hfId)
                 else compare `on` hfRefLvl

instance Outputable HoleFit where
    ppr = pprHoleFit debugHoleFitDispConfig

instance (HasOccName a, HasOccName b) => HasOccName (Either a b) where
    occName = either occName occName

instance HasOccName GlobalRdrElt where
    occName = occName . gre_name


-- For pretty printing hole fits, we display the name and type of the fit,
-- with added '_' to represent any extra arguments in case of a non-zero
-- refinement level.
pprHoleFit :: HoleFitDispConfig -> HoleFit -> SDoc
pprHoleFit (HFDC sWrp sWrpVars sTy sProv sMs) hf = hang display 2 provenance
    where name = case hfElem hf of
                      Just gre -> gre_name gre
                      Nothing -> idName (hfId hf)
          ty = hfType hf
          matches =  hfMatches hf
          wrap = hfWrap hf
          tyApp = sep $ map ((text "@" <>) . pprParendType) wrap
          tyAppVars = sep $ punctuate comma $
              map (\(v,t) -> ppr v <+> text "~" <+> pprParendType t) $
                zip vars wrap
            where
              vars = unwrapTypeVars ty
              -- Attempts to get all the quantified type variables in a type,
              -- e.g.
              -- return :: forall (m :: * -> *) Monad m => (forall a . a) -> m a
              -- into [m, a]
              unwrapTypeVars :: Type -> [TyVar]
              unwrapTypeVars t = vars ++ case splitFunTy_maybe unforalled of
                                   Just (_, unfunned) -> unwrapTypeVars unfunned
                                   _ -> []
                where (vars, unforalled) = splitForAllTys t
          holeVs = sep $ map (parens . (text "_" <+> dcolon <+>) . ppr) matches
          holeDisp = if sMs then holeVs
                     else sep $ replicate (length matches) $ text "_"
          occDisp = pprPrefixOcc name
          tyDisp = ppWhen sTy $ dcolon <+> ppr ty
          has = not . null
          wrapDisp = ppWhen (has wrap && (sWrp || sWrpVars))
                      $ text "with" <+> if sWrp || not sTy
                                        then occDisp <+> tyApp
                                        else tyAppVars
          funcInfo = ppWhen (has matches && sTy) $
                       text "where" <+> occDisp <+> tyDisp
          subDisp = occDisp <+> if has matches then holeDisp else tyDisp
          display =  subDisp $$ nest 2 (funcInfo $+$ wrapDisp)
          provenance = ppWhen sProv $
            parens $
                case hfElem hf of
                    Just gre -> pprNameProvenance gre
                    Nothing -> text "bound at" <+> ppr (getSrcLoc name)

getLocalBindings :: TidyEnv -> Ct -> TcM [Id]
getLocalBindings tidy_orig ct
 = do { (env1, _) <- zonkTidyOrigin tidy_orig (ctLocOrigin loc)
      ; go env1 [] (removeBindingShadowing $ tcl_bndrs lcl_env) }
  where
    loc     = ctEvLoc (ctEvidence ct)
    lcl_env = ctLocEnv loc

    go :: TidyEnv -> [Id] -> [TcBinder] -> TcM [Id]
    go _ sofar [] = return (reverse sofar)
    go env sofar (tc_bndr : tc_bndrs) =
        case tc_bndr of
          TcIdBndr id _ -> keep_it id
          _ -> discard_it
     where
        discard_it = go env sofar tc_bndrs
        keep_it id = go env (id:sofar) tc_bndrs

-- See Note [Valid hole fits include ...]
findValidHoleFits :: TidyEnv        --The tidy_env for zonking
                  -> [Implication]  --Enclosing implications for givens
                  -> [Ct] -- The  unsolved simple constraints in the
                          -- implication for the hole.
                  -> Ct   -- The hole constraint itself
                  -> TcM (TidyEnv, SDoc)
findValidHoleFits tidy_env implics simples ct | isExprHoleCt ct =
  do { rdr_env <- getGlobalRdrEnv
     ; lclBinds <- getLocalBindings tidy_env ct
     ; maxVSubs <- maxValidHoleFits <$> getDynFlags
     ; hfdc <- getHoleFitDispConfig
     ; sortingAlg <- getSortingAlg
     ; refLevel <- refLevelHoleFits <$> getDynFlags
     ; traceTc "findingValidHoleFitsFor { " $ ppr ct
     ; traceTc "hole_lvl is:" $ ppr hole_lvl
     ; traceTc "implics are: " $ ppr implics
     ; traceTc "simples are: " $ ppr simples
     ; traceTc "locals are: " $ ppr lclBinds
     ; let (lcl, gbl) = partition gre_lcl (globalRdrEnvElts rdr_env)
           -- We remove binding shadowings here, but only for the local level.
           -- this is so we e.g. suggest the global fmap from the Functor class
           -- even though there is a local definition as well, such as in the
           -- Free monad example.
           locals = removeBindingShadowing $ map Left lclBinds ++ map Right lcl
           globals = map Right gbl
           -- If the type is an unbound type variable with no relevant
           -- constraints, we'll (correctly) match anything in scope. However,
           -- this is not very useful, so in that case we only check the local
           -- bindings.
           to_check = if null relevantCts && isTyVarTy hole_ty
                      then locals else locals ++ globals
     ; (searchDiscards, subs) <-
        findSubs sortingAlg maxVSubs to_check (hole_ty, [])
     ; (tidy_env, tidy_subs) <- zonkSubs tidy_env subs
     ; tidy_sorted_subs <- sortFits sortingAlg tidy_subs
     ; let (pVDisc, limited_subs) = possiblyDiscard maxVSubs tidy_sorted_subs
           vDiscards = pVDisc || searchDiscards
     ; let vMsg = ppUnless (null limited_subs) $
                    hang (text "Valid hole fits include") 2 $
                      vcat (map (pprHoleFit hfdc) limited_subs)
                        $$ ppWhen vDiscards subsDiscardMsg
     -- Refinement hole fits. See Note [Valid refinement hole fits include ...]
     ; (tidy_env, refMsg) <- if refLevel >= Just 0 then
         do { maxRSubs <- maxRefHoleFits <$> getDynFlags
            -- We can use from just, since we know that Nothing >= _ is False.
            ; let refLvls = [1..(fromJust refLevel)]
            -- We make a new refinement type for each level of refinement, where
            -- the level of refinement indicates number of additional arguments
            -- to allow.
            ; ref_tys <- mapM mkRefTy refLvls
            ; traceTc "ref_tys are" $ ppr ref_tys
            ; refDs <- mapM (findSubs sortingAlg maxRSubs to_check) ref_tys
            ; (tidy_env, tidy_rsubs) <- zonkSubs tidy_env $ concatMap snd refDs
            ; tidy_sorted_rsubs <- sortFits sortingAlg tidy_rsubs
            -- For refinement substitutions we want matches
            -- like id (_ :: t), head (_ :: [t]), asTypeOf (_ :: t),
            -- and others in that vein to appear last, since these are
            -- unlikely to be the most relevant fits.
            ; (tidy_env, tidy_hole_ty) <- zonkTidyTcType tidy_env hole_ty
            ; let hasExactApp = any (tcEqType tidy_hole_ty) . hfWrap
                  (exact, not_exact) = partition hasExactApp tidy_sorted_rsubs
                  (pRDisc, exact_last_rfits) =
                    possiblyDiscard maxRSubs $ not_exact ++ exact
                  rDiscards = pRDisc || any fst refDs
            ; return (tidy_env,
                ppUnless (null tidy_sorted_rsubs) $
                  hang (text "Valid refinement hole fits include") 2 $
                  vcat (map (pprHoleFit hfdc) exact_last_rfits)
                    $$ ppWhen rDiscards refSubsDiscardMsg) }
       else return (tidy_env, empty)
     ; traceTc "findingValidHoleFitsFor }" empty
     ; return (tidy_env, vMsg $$ refMsg) }
  where
    -- We extract the type, the tcLevel and the types free variables
    -- from from the constraint.
    hole_ty :: TcPredType
    hole_ty = ctPred ct
    hole_fvs = tyCoFVsOfType hole_ty
    hole_lvl = ctLocLevel $ ctEvLoc $ ctEvidence ct

    -- We make a refinement type by adding a new type variable in front
    -- of the type of t h hole, going from e.g. [Integer] -> Integer
    -- to t_a1/m[tau:1] -> [Integer] -> Integer. This allows the simplifier
    -- to unify the new type variable with any type, allowing us
    -- to suggest a "refinement hole fit", like `(foldl1 _)` instead
    -- of only concrete hole fits like `sum`.
    mkRefTy :: Int -> TcM (TcType, [TcTyVar])
```

```
mkRefTy refLvl = (wrapWithVars &&& id) <$> newTyVars
  where newTyVars = replicateM refLvl $ setLvl <$>
                        (newOpenTypeKind >>= newFlexiTyVar)
        setLvl = flip setMetaTyVarTcLevel hole_lvl
        wrapWithVars vars = mkFunTys (map mkTyVarTy vars) hole_ty

sortFits :: SortingAlg    -- How we should sort the hole fits
         -> [HoleFit]     -- The subs to sort
         -> TcM [HoleFit]
sortFits NoSorting subs = return subs
sortFits BySize subs
    = (++) <$> sortBySize (sort lclFits)
           <*> sortBySize (sort gblFits)
    where (lclFits, gblFits) = span isLocalHoleFit subs

-- To sort by subsumption, we invoke the sortByGraph function, which
-- builds the subsumption graph for the fits and then sorts them using a
-- graph sort.  Since we want locals to come first anyway, we can sort
-- them separately. The substitutions are already checked in local then
-- global order, so we can get away with using span here.
-- We use (<*>) to expose the parallelism, in case it becomes useful later.
sortFits BySubsumption subs
    = (++) <$> sortByGraph (sort lclFits)
           <*> sortByGraph (sort gblFits)
    where (lclFits, gblFits) = span isLocalHoleFit subs

isLocalHoleFit :: HoleFit -> Bool
isLocalHoleFit hf = case hfElem hf of
                      Just gre -> gre_lcl gre
                      Nothing -> True

-- See Note [Relevant Constraints]
relevantCts :: [Ct]
relevantCts = if isEmptyVarSet (fvVarSet hole_fvs) then []
              else filter isRelevant simples
  where ctFreeVarSet :: Ct -> VarSet
        ctFreeVarSet = fvVarSet . tyCoFVsOfType . ctPred
        hole_fv_set = fvVarSet hole_fvs
        anyFVMentioned :: Ct -> Bool
        anyFVMentioned ct = not $ isEmptyVarSet $
                              ctFreeVarSet ct `intersectVarSet` hole_fv_set
        -- We filter out those constraints that have no variables (since
        -- they won't be solved by finding a type for the type variable
        -- representing the hole) and also other holes, since we're not
        -- trying to find hole fits for many holes at once.
        isRelevant ct = not (isEmptyVarSet (ctFreeVarSet ct))
                        && anyFVMentioned ct
                        && not (isHoleCt ct)

unfoldWrapper :: HsWrapper -> [Type]
unfoldWrapper = reverse . unfWrp'
  where unfWrp' (WpTyApp ty) = [ty]
        unfWrp' (WpCompose w1 w2) = unfWrp' w1 ++ unfWrp' w2
        unfWrp' _ = []
```

```haskell
-- We only clone flexi type variables, and we need to be able to check
-- whether a variable is filled or not.
isFlexiTyVar :: TcTyVar -> TcM Bool
isFlexiTyVar tv | isMetaTyVar tv = isFlexi <$> readMetaTyVar tv
isFlexiTyVar _ = return False

-- Takes a list of free variables and makes sure that the given action
-- is run with the right TcLevel and restores any Flexi type
-- variables after the action is run.
withoutUnification :: FV -> TcM a -> TcM a
withoutUnification free_vars action
  = do { flexis <- filterM isFlexiTyVar fuvs
        ; result <- setTcLevel deepestFreeTyVarLvl action
          -- Reset any mutated free variables
        ; mapM_ restore flexis
        ; return result }
  where restore = flip writeTcRef Flexi . metaTyVarRef
        fuvs = fvVarList free_vars
        deepestFreeTyVarLvl = foldl' max topTcLevel $ map tcTyVarLevel fuvs

-- The real work happens here, where we invoke the type checker using
-- tcCheckHoleFit to see whether the given type fits the hole.
fitsHole :: (TcType, [TcTyVar]) -- The type of the hole wrapped with the
                                -- refinement variables created to simulate
                                -- additional holes (if any), and the list
                                -- of those variables (possibly empty).
                                -- As an example: If the actual type of the
                                -- hole (as specified by the hole
                                -- constraint CHoleExpr passed to
                                -- findValidHoleFits) is t and we want to
                                -- simulate N additional holes, h_ty will
                                -- be  r_1 -> ... -> r_N -> t, and
                                -- ref_vars will be [r_1, ... , r_N].
                                -- In the base case with no additional
                                -- holes, h_ty will just be t and ref_vars
                                -- will be [].
         -> TcType -- The type we're checking to whether it can be
                   -- instantiated to the type h_ty.
         -> TcM (Maybe ([TcType], [TcType])) -- If it is not a match, we
                                             -- return Nothing. Otherwise,
                                             -- we Just return the list of
                                             -- types that quantified type
                                             -- variables in ty would take
                                             -- if used in place of h_ty,
                                             -- and the list types of any
                                             -- additional holes simulated
                                             -- with the refinement
                                             -- variables in ref_vars.
fitsHole (h_ty, ref_vars) ty =
-- We wrap this with the withoutUnification to avoid having side-effects
-- beyond the check, but we rely on the side-effects when looking for
-- refinement hole fits, so we can't wrap the side-effects deeper than this.
  withoutUnification fvs $
  do { traceTc "checkingFitOf {" $ ppr ty
     ; (fits, wrp) <- tcCheckHoleFit (listToBag relevantCts) implics h_ty ty
     ; traceTc "Did it fit?" $ ppr fits
```

```
        ; traceTc "wrap is: " $ ppr wrp
        ; traceTc "checkingFitOf }" empty
        ; z_wrp_tys <- zonkTcTypes (unfoldWrapper wrp)
        -- We'd like to avoid refinement suggestions like `id _ _` or
        -- `head _ _`, and only suggest refinements where our all phantom
        -- variables got unified during the checking. This can be disabled
        -- with the `-fabstract-refinement-hole-fits` flag.
        -- Here we do the additional handling when there are refinement
        -- variables, i.e. zonk them to read their final value to check for
        -- abstract refinements, and to report what the type of the simulated
        -- holes must be for this to be a match.
        ; if fits
          then if null ref_vars
               then return (Just (z_wrp_tys, []))
               else do { let -- To be concrete matches, matches have to
                             -- be more than just an invented type variable.
                             fvSet = fvVarSet fvs
                             notAbstract :: TcType -> Bool
                             notAbstract t = case getTyVar_maybe t of
                                               Just tv -> tv `elemVarSet` fvSet
                                               _ -> True
                             allConcrete = all notAbstract z_wrp_tys
                       ; z_vars  <- zonkTcTyVars ref_vars
                       ; let z_mtvs = mapMaybe tcGetTyVar_maybe z_vars
                       ; allFilled <- not <$> anyM isFlexiTyVar z_mtvs
                       ; allowAbstract <- goptM Opt_AbstractRefHoleFits
                       ; if allowAbstract || (allFilled && allConcrete )
                         then return $ Just (z_wrp_tys, z_vars)
                         else return Nothing }
          else return Nothing }
     where fvs = mkFVs ref_vars `unionFV` hole_fvs `unionFV` tyCoFVsOfType ty

    -- We zonk the hole fits so that the output aligns with the rest
    -- of the typed hole error message output.
    zonkSubs :: TidyEnv -> [HoleFit] -> TcM (TidyEnv, [HoleFit])
    zonkSubs = zonkSubs' []
      where zonkSubs' zs env [] = return (env, reverse zs)
            zonkSubs' zs env (hf:hfs) = do { (env', z) <- zonkSub env hf
                                           ; zonkSubs' (z:zs) env' hfs }
            zonkSub env hf@HoleFit{hfType = ty, hfMatches = m, hfWrap = wrp}
              = do { (env, ty') <- zonkTidyTcType env ty
                   ; (env, m') <- zonkTidyTcTypes env m
                   ; (env, wrp') <- zonkTidyTcTypes env wrp
                   ; let zFit = hf {hfType = ty', hfMatches = m', hfWrap = wrp'}
                   ; return (env, zFit ) }

    -- Based on the flags, we might possibly discard some or all the
    -- fits we've found.
    possiblyDiscard :: Maybe Int -> [HoleFit] -> (Bool, [HoleFit])
    possiblyDiscard (Just max) fits = (fits `lengthExceeds` max, take max fits)
    possiblyDiscard Nothing fits = (False, fits)


    -- Sort by size uses as a measure for relevance the sizes of the
    -- different types needed to instantiate the fit to the type of the hole.
    -- This is much quicker than sorting by subsumption, and gives reasonable
```

```
-- results in most cases.
sortBySize :: [HoleFit] -> TcM [HoleFit]
sortBySize = return . sortOn sizeOfFit
  where sizeOfFit :: HoleFit -> TypeSize
        sizeOfFit = sizeTypes . nubBy tcEqType .  hfWrap

-- Based on a suggestion by phadej on #ghc, we can sort the found fits
-- by constructing a subsumption graph, and then do a topological sort of
-- the graph. This makes the most specific types appear first, which are
-- probably those most relevant. This takes a lot of work (but results in
-- much more useful output), and can be disabled by
-- '-fno-sort-valid-hole-fits'.
sortByGraph :: [HoleFit] -> TcM [HoleFit]
sortByGraph fits = go [] fits
  where tcSubsumesWCloning :: TcType -> TcType -> TcM Bool
        tcSubsumesWCloning ht ty = withoutUnification fvs (tcSubsumes ht ty)
          where fvs = tyCoFVsOfTypes [ht,ty]
        go :: [(HoleFit, [HoleFit])] -> [HoleFit] -> TcM [HoleFit]
        go sofar [] = do { traceTc "subsumptionGraph was" $ ppr sofar
                         ; return $ uncurry (++)
                                  $ partition isLocalHoleFit topSorted }
          where toV (hf, adjs) = (hf, hfId hf, map hfId adjs)
                (graph, fromV, _) = graphFromEdges $ map toV sofar
                topSorted = map ((\(h,_,_) -> h) . fromV) $ topSort graph
        go sofar (hf:hfs) =
          do { adjs <-
                 filterM (tcSubsumesWCloning (hfType hf) . hfType) fits
             ; go ((hf, adjs):sofar) hfs }

findSubs :: SortingAlg         -- Whether we should sort the subs or not
         -> Maybe Int          -- How many we should output, if limited
         -> [Either Id GlobalRdrElt] -- The elements to check whether fit
         -> (TcType, [TcTyVar]) -- The type to check for fits and refinement
                                -- variables for emulating additional holes
         -> TcM (Bool, [HoleFit]) -- We return whether or not we stopped due
                                  -- to running out of gas and the fits we
                                  -- found.
-- We don't check if no output is desired.
findSubs _ (Just 0) _ _ = return (False, [])
findSubs sortAlg maxSubs to_check ht@(hole_ty, _) =
  do { traceTc "checkingFitsFor {" $ ppr hole_ty
     -- If we're not going to sort anyway, we can stop going after
     -- having found `maxSubs` hole fits.
     ; let limit = if sortAlg > NoSorting then Nothing else maxSubs
     ; (discards, subs) <- go [] emptyVarSet limit ht to_check
     ; traceTc "checkingFitsFor }" empty
     ; return (discards, subs) }
  where
    -- Kickoff the checking of the elements.
    -- We iterate over the elements, checking each one in turn for whether
    -- it fits, and adding it to the results if it does.
    go :: [HoleFit]            -- What we've found so far.
       -> VarSet               -- Ids we've already checked
       -> Maybe Int            -- How many we're allowed to find, if limited
       -> (TcType, [TcTyVar]) -- The type, and its refinement variables.
       -> [Either Id GlobalRdrElt] -- The elements we've yet to check.
```

```
            -> TcM (Bool, [HoleFit])
        go subs _ _ _ [] = return (False, reverse subs)
        go subs _ (Just 0) _ _ = return (True, reverse subs)
        go subs seen maxleft ty (el:elts) =
          do { traceTc "lookingUp" $ ppr el
             ; maybeThing <- lookup el
             ; case maybeThing of
                 Just id | not_trivial id ->
                           do { fits <- fitsHole ty (idType id)
                              ; case fits of
                                  Just (wrp, matches) -> keep_it id wrp matches
                                  _ -> discard_it }
                 _ -> discard_it }
          where discard_it = go subs seen maxleft ty elts
                keep_it id wrp ms = go (fit:subs) (extendVarSet seen id)
                                  ((\n -> n - 1) <$> maxleft) ty elts
                  where fit = HoleFit { hfElem = mbel , hfId = id
                                      , hfType = idType id
                                      , hfRefLvl = length (snd ty)
                                      , hfWrap = wrp , hfMatches = ms }
                        mbel = either (const Nothing) Just el
                -- We want to filter out undefined and the likes from GHC.Err
                not_trivial id = nameModule_maybe (idName id) /= Just gHC_ERR
                lookup :: Either Id GlobalRdrElt -> TcM (Maybe Id)
                lookup (Left id) = return $ Just id
                lookup (Right el) =
                  do { thing <- tcLookup (gre_name el)
                     ; case thing of
                         ATcId {tct_id = id} -> return $ Just id
                         AGlobal (AnId id)   -> return $ Just id
                         AGlobal (AConLike (RealDataCon con))  ->
                           return $ Just (dataConWrapId con)
                         _ -> return Nothing }


-- We don't (as of yet) handle holes in types, only in expressions.
findValidHoleFits env _ _ _ = return (env, empty)

subsDiscardMsg :: SDoc
subsDiscardMsg =
    text "(Some hole fits suppressed;" <+>
    text "use -fmax-valid-hole-fits=N" <+>
    text "or -fno-max-valid-hole-fits)"

refSubsDiscardMsg :: SDoc
refSubsDiscardMsg =
    text "(Some refinement hole fits suppressed;" <+>
    text "use -fmax-refinement-hole-fits=N" <+>
    text "or -fno-max-refinement-hole-fits)"


-- | Reports whether first type (ty_a) subsumes the second type (ty_b),
-- discarding any errors. Subsumption here means that the ty_b can fit into the
-- ty_a, i.e. `tcSubsumes a b == True` if b is a subtype of a.
tcSubsumes :: TcSigmaType -> TcSigmaType -> TcM Bool
tcSubsumes ty_a ty_b = fst <$> tcCheckHoleFit emptyBag [] ty_a ty_b
```

```haskell
-- | A tcSubsumes which takes into account relevant constraints, to fix trac
-- #14273. This makes sure that when checking whether a type fits the hole,
-- the type has to be subsumed by type of the hole as well as fulfill all
-- constraints on the type of the hole.
-- Note: The simplifier may perform unification, so make sure to restore any
-- free type variables to avoid side-effects.
tcCheckHoleFit :: Cts                   -- Any relevant Cts to the hole.
               -> [Implication]         -- The nested implications of the hole
               -> TcSigmaType           -- The type of the hole.
               -> TcSigmaType           -- The type to check whether fits.
               -> TcM (Bool, HsWrapper)
tcCheckHoleFit _ _ hole_ty ty | hole_ty `eqType` ty
    = return (True, idHsWrapper)
tcCheckHoleFit relevantCts implics hole_ty ty = discardErrs $
  do { (wrp, wanted) <- captureConstraints $ tcSubType_NC ExprSigCtxt ty hole_ty
     ; traceTc "Checking hole fit {" empty
     ; traceTc "wanteds are: " $ ppr wanted
     -- We add the relevantCts to the wanteds generated by the call to
     -- tcSubType_NC, see Note [Relevant Constraints]
     ; let w_rel_cts = addSimples wanted relevantCts
     ; if isEmptyWC w_rel_cts
       then traceTc "}" empty >> return (True, wrp)
       else do { fresh_binds <- newTcEvBinds
                 -- We wrap the WC in the nested implications, see
                 -- Note [Nested Implications]
               ; let outermost_first = reverse implics
                     setWC = setWCAndBinds fresh_binds
                     w_givens = foldr setWC w_rel_cts outermost_first
               ; traceTc "w_givens are: " $ ppr w_givens
               ; rem <- runTcSDeriveds $ simpl_top w_givens
               -- We don't want any insoluble or simple constraints left, but
               -- solved implications are ok (and neccessary for e.g. undefined)
               ; traceTc "rems was:" $ ppr rem
               ; traceTc "}" empty
               ; return (isSolvedWC rem, wrp) } }
     where
       setWCAndBinds :: EvBindsVar         -- Fresh ev binds var.
                     -> Implication        -- The implication to put WC in.
                     -> WantedConstraints  -- The WC constraints to put implic.
                     -> WantedConstraints  -- The new constraints.
       setWCAndBinds binds imp wc
         = WC { wc_simple = emptyBag
              , wc_impl = unitBag $ imp { ic_wanted = wc , ic_binds = binds } }
```

A.2 The Sorting Library

A.2.1 ONotation

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

```
import GHC.TypeLits
import Data.Type.Bool
import Data.Type.Equality


-- We define a very simplistic O notation, with sufficient expressiveness
-- to capture the complexity of a few simple sorting algorithms
data AsymptoticPoly = 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 :: AsymptoticPoly) = a

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

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

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

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

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

infix 4 >=.
infixl 7 *., ^.
```

A.2.2 The Sorting Library

```
{-# LANGUAGE TypeInType, TypeOperators, TypeFamilies, ConstraintKinds #-}

module Sorting ( mergeSort, heapSort
               , quickSort, insertionSort
               , Sorted, sortedBy) where

import Data.Type.Bool
import ONotation
import Data.List (sort)


-- Stable sorts must be stable, but unstable can be, but don't need to.
type IsStable s = (s || True) ~ True
-- We encode in the return type of the sorting function its average complexity,
-- memory use and stability.
newtype Sorted (cpu :: AsymptoticPoly) -- The minimum operational complexity
                                       -- this algorithm satisfies.
               (mem :: AsymptoticPoly) -- The minimum space complexity this
                                       -- algorithm satisfies.
```

```haskell
               (stable :: Bool)         -- Whether the sort is stable or not.
               a                        -- What was being sorted.
               = Sorted {sortedBy :: [a]}

-- Merge sort is O(N*Log(N)) on average in complexity, so that's the
-- minimum complexity we promise to satisfy. Same goes with memory, which is
-- O(N), and as we all know, mergesort is a stable sorting algoritm.
mergeSort :: (Ord a, n >=. O(N*.LogN), m >=. O(N), IsStable s)
          => [a] -> Sorted n m s a
mergeSort = Sorted . sort

insertionSort :: (Ord a, n >=. O(N^.2), m >=. O(One), IsStable s)
              => [a] -> Sorted n m s a
insertionSort = Sorted . sort

-- Note that we don't actually check the complexity (as evidenced by them all
-- being implemented with sort, a smooth applicative merge sort). With more
-- dependent types however, some of these properties might be verifiable.
quickSort :: (Ord a, n >=. O(N*.LogN), m >=. O(N)) => [a] -> Sorted n m False a
quickSort = Sorted . sort

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

A.3 DCC

A.3.1 The DCC Library

The following implementation of DCC is taken from the "Encoding DCC in Haskell" paper [30]:

```haskell
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE TypeFamilies  #-}
{-# LANGUAGE DataKinds     #-}

module Lattice where

-- Two point lattice
data Lattice = H | L deriving (Show, Eq)

type family l <= h where
  H <= L = False
  _ <= _ = True
{-# LANGUAGE TypeOperators         #-}
{-# LANGUAGE TypeFamilies          #-}
{-# LANGUAGE DataKinds             #-}
{-# LANGUAGE FlexibleInstances     #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE UndecidableInstances  #-}
{-# LANGUAGE CPP #-}

module DCC where

import Control.Monad
import Control.Monad.Trans
import Data.Type.Bool

import Lattice

# if !MIN_VERSION_base(4,11,0)
import Data.Semigroup ( Semigroup, (<>) )
# endif

-- The ‘T‘ monad family from the DCC paper
newtype T (l :: Lattice) a = T { unT :: a }

{- Notion of protected at -}
type family ProtectedAt (t :: *) (l :: Lattice) where
  ProtectedAt (T l’ a) l = l <= l’

  -- This diverges from the original DCC calculus,
  -- this is not a problem as () can not transmit any
  -- information
  ProtectedAt ()       l = True

  -- This line requires ‘UndecidableInstances‘, but since the type family is closed
  -- and _obviously_ total, we don’t need to worry!
  ProtectedAt (s, t) l = (ProtectedAt s l) && (ProtectedAt t l)
  ProtectedAt (s -> t) l = ProtectedAt t l
  ProtectedAt s l = False

-- The bind from the DCC paper
infixl >>>=

-- Superbind
(>>>=) :: (s ‘ProtectedAt‘ l) ~ True => T l a -> (a -> s) -> s
t >>>= f = f (unT t)

-- T is most surely a monad
instance Monad (T l) where
  return = T
  (>>=)   = (>>>=)

instance Applicative (T l) where
  pure   = return
  (<*>) = ap

instance Functor (T l) where
  fmap = liftM

class MonadT mt l where
  liftT :: T l a -> mt a

instance MonadT (T l) l where
  liftT = id

instance (MonadT inner l, Monad inner, MonadTrans mt) => MonadT (mt inner) l where
  liftT = lift . liftT

instance Semigroup m => Semigroup (T l m) where
  (<>) = liftM2 (<>)
```

```
instance Monoid m => Monoid (T l m) where
  mempty = return mempty
# if !MIN_VERSION_base(4,11,0)
  mappend = liftM2 mappend
# endif
```

A.3.2 The Restaurant Info Library

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE Safe #-}
module RestaurantInfo ( Lattice(..), T -- We don't export the constructors for T
                                       -- since then users could use unT at will.
                      , User(..) , Loc , Restaurant
                      , getUserInfo, readPublic
                      , bestNearbyRestaurant, isInGothenburg, isAllowedToDrink
                      ) where

import DCC
import Lattice
import Data.List ( sortOn )
import Data.Maybe ( listToMaybe )


type Loc = (Int, Int)
data Restaurant = Restaurant { name :: String
                             , coords :: Loc
                             , rating :: Int } deriving (Show)

data User = User { firstName :: String
                 , lastName :: String
                 , location :: Loc
                 , age :: Int } deriving (Show)



lookupRestaurants :: Loc -> [Restaurant]
lookupRestaurants (0,0) = [ Restaurant "Bhoga" (0,0) 3
                          , Restaurant "Koka" (1,2) 1]
lookupRestaurants _ = []

trust :: (a -> b) -> T H a -> T L b
trust f = T . f . unT

-- Exported functions:

bestNearbyRestaurant :: T H User -> T L (Maybe Restaurant)
bestNearbyRestaurant = trust (listToMaybe . reverse . sortOn rating
                             . lookupRestaurants . location)

isInGothenburg :: T H User -> T L Bool
isInGothenburg = trust (inGothenburg . location)
  where inGothenburg (x,y) = (x >= -5) && (x <= 5) && (y >= -5) && (y <= 5)

isAllowedToDrink :: T H User -> T L Bool
```

```haskell
isAllowedToDrink = trust ((>= 20) . age)

-- We allow reading of public information
readPublic :: T L a -> a
readPublic = unT

-- Info on the user can be read, but only in a private
-- environment.
getUserInfo :: IO (T H User)
getUserInfo = (pure . pure) $ User "Matthias Pall" "Gissurarson" (0,0) 26
```

A.4 The PiSkjerm Project

A.4.1 Screen Handling

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE CPP #-}
module InkyPhat ( runInky, runInkyLoop
                  -- commands
                , text, paste, display, setRotation, clear

                -- Data
                , image, size, dimensions, readString
                -- Types
                , Color (..) , Font (..), Image
                , InkyException (..), InkyIO
    ) where


import System.Process
import Control.Concurrent
import GHC.IO.Handle (Handle, hClose, hFlush, hFlushAll, hWaitForInput)
import Data.Maybe (fromMaybe)
import Control.Monad (join)

import Control.Monad.Reader
import Control.Monad.Except
import Data.Text
import qualified Data.Text as T

import Prelude hiding ( hPutStrLn, hGetLine )
import Data.Text.IO

import Text.Read (readEither)

data InkyException = ValueParseError Text
                   | PythonError Text
                   | NoDataError deriving (Show, Eq)

type InkyIO = ExceptT InkyException (ReaderT (Handle, Handle, Handle) IO)

-- To avoid issues with FFI, we'll just go the easy
-- interpeter way.
sendCommand :: Text -> InkyIO ()
sendCommand cmd =
    do (stdin, _, _) <- ask
       dlog cmd
       liftIO $ do hPutStrLn stdin cmd
                   hFlushAll stdin

dlog :: Text -> InkyIO ()
# if debug
dlog = putStrLn
# else
dlog _ = return ()
#endif

readValue :: Read a => Text -> InkyIO a
readValue cmd  = do val <- readEither . unpack <$> readString cmd
                    case val of
                      Left err -> throwError (ValueParseError $ pack err)
                      Right v -> return v


readString :: Text -> InkyIO Text
readString cmd =
  do (stdin, stdout, stderr) <- ask
     liftIO $ hFlushAll stdout
     sendCommand $ "print(" <> cmd <> ")"
     --- We wait at max 30 seconds for output
     val <- liftIO $ readWithTimeout stdout (30*1000)
     case val of
        Just v -> do dlog v
                     return v
        _ -> do err <- liftIO $ readWithTimeout stderr (5 * 1000)
                case err of
                  Just e -> throwError (PythonError e)
                  _ -> throwError NoDataError
 where
    readWithTimeout :: Handle -> Int -> IO (Maybe Text)
    readWithTimeout handle timeout
     = do hasOutput <- hWaitForInput handle timeout
          if hasOutput then Just <$> hGetLine handle
                       else return Nothing
initialCommands :: [Text]
initialCommands
    = [ "import inkyphat"
      , "from urllib import request"
      , "from datetime import datetime"]

data Color = Black | White | Red deriving (Show)

class InkyPhatVal a where
 toIPVal :: a -> Text

instance InkyPhatVal Color where
  toIPVal Black = "inkyphat.BLACK"
  toIPVal White = "inkyphat.WHITE"
  toIPVal Red   = "inkyphat.RED"
```

```
-- We only have the one font for now
newtype Font = Font Int

instance InkyPhatVal Font where
  toIPVal (Font a) =
    "inkyphat.ImageFont.truetype(inkyphat.fonts.PressStart2P,"
      <> pack (show a) <>")"

display :: InkyIO ()
display = sendCommand "inkyphat.show()"

clear :: InkyIO ()
clear = sendCommand "inkyphat.clear()"

type Image = Text

setRotation :: Int -> InkyIO ()
setRotation rot =
  sendCommand $ "inkyphat.set_rotation(" <> pack (show rot) <> ")"

image :: Text -> Text -> InkyIO Image
image name loc =
  do sendCommand $ name <> " = inkyphat.Image.open('" <> loc <> "')"
     return name


paste :: Image -> (Int, Int) -> InkyIO ()
paste img loc =
  sendCommand $ intercalate ","  ["inkyphat.paste(" <> img
                                 , pack (show loc) <> ")"]

text :: (Int, Int) -> Text -> Maybe Color -> Maybe Font -> InkyIO ()
text xy text color font =
    sendCommand $ intercalate "," ["inkyphat.text(" <> pack (show xy)
                                  , "'" <> text <> "'"
                                  , toIPVal col
                                  , toIPVal fon <> ")" ]
  where fon = fromMaybe (Font 12) font
        col =  fromMaybe Black color

size :: Image -> InkyIO (Int, Int)
size img = readValue $ "(" <> img <> ".size[0], " <> img <> ".size[1])"

dimensions :: InkyIO (Int, Int)
dimensions = readValue "(inkyphat.WIDTH, inkyphat.HEIGHT)"

runInkyLoop :: InkyIO a -> IO ()
runInkyLoop loop
 = do r <- runInky loop
      case r of
        Left err -> print err
        _ -> return ()
      runInkyLoop loop
```

```haskell
runInky :: InkyIO a -> IO (Either InkyException a)
runInky action =
  do (Just stdin, Just stdout , Just stderr, proc)
        <- createProcess $ cp { std_in = CreatePipe, std_out = CreatePipe
                              , std_err = CreatePipe }
     r <- flip runReaderT (stdin, stdout, stderr) $
            runExceptT $
              do mapM_ sendCommand initialCommands
                 action
     terminateProcess proc
     mapM_ hClose [stdin, stdout, stderr]
     return r
  where cp = shell "python3 -i"
```

A.4.2 Types and Functions for Bus Lines

```haskell
{-# LANGUAGE OverloadedStrings, DeriveAnyClass, DeriveGeneric,
             TypeSynonymInstances, DuplicateRecordFields #-}
module BusTimes where

import System.Environment
import Data.Aeson
import Data.Aeson.Types

import GHC.Generics (Generic)

import Data.List (nub)

import qualified Data.ByteString.Base64 as B64
import Data.ByteString.Char8 (ByteString, pack, unpack)

getAuthCredentials :: IO (String, String)
-- First we have to read the secrets from the environment,
-- so we add System.Environment to the import list and try
-- getSecret = _
-- one of he matches is the getEnv function!
-- Let's use that and get the key and secret
-- getAuthCredentials = do key <- _b "VASTTRAFIKKEY"
--                         secret <- _a "VASTTRAFIKSECRET"
--                         return (key, secret)
--
-- One of the matches is getEnv, which sounds like what
-- we want. Let's try that:
getAuthCredentials = do key <- getEnv "VASTTRAFIKKEY"
                        secret <- getEnv "VASTTRAFIKSECRET"
                        return (key, secret)
-- It works!

-- The token is the base64 encoded string "key:secret",
authToken :: String -> String -> String
authToken key secret = encodeToken $ key ++ ":" ++ secret
  where
    -- We need to base64 encode this authorization,
    -- so let's use the base64-bytesting package.
    encodeToken :: String -> String
    encodeToken token = result
      where
        -- First we need to get to bytestring
        toBS :: String -> ByteString
        -- toBs = _
        -- pack has the right type and sounds reasonable, let's use that!
        toBS = pack
        -- Next we need to turn the bytestring into base64 string
        b64ed :: ByteString
        -- b64ed = _ $ toBs token
        -- B64.encode sounds like what we need!
        b64ed = B64.encode $ toBS token
        -- And finally, we need to turn it back into a string
        result :: String
        -- result = _ b64ed
        -- unpack should to the trick!
        result = unpack b64ed

-- Now, let's try to fetch a token!

data TokenResponse = TR { scope :: String
                        , token_type :: String
                        , expires_in:: Int
                        , access_token :: String
                        } deriving (Show, Generic, FromJSON)

data BusResponse = BR { departureBoard :: DepartureBoard} deriving Show

instance FromJSON BusResponse where
  parseJSON = withObject "BusResponse" $ \v -> BR <$> v .: "DepartureBoard"

data DepartureBoard = DB { departure :: [Departure] } deriving Show

instance FromJSON DepartureBoard where
  parseJSON = withObject "DepartureBoard" $ \v -> DB <$> v .:? "Departure" .!= []


data Departure = DP { direction :: String
                    , track :: String
                    , sname :: String
                    , rtTime :: Maybe String
                    , rtDate :: Maybe String
                    , time :: String
                    , date :: String
                    } deriving (Show, Generic, FromJSON)

type Token = String
type BusStop = Integer

data BusLine = BL { name :: String
                  , track :: String
                  , departures :: [String] } deriving (Show, Eq)

getLines :: BusResponse -> [BusLine]
getLines (BR (DB dep)) = concatMap getLine names
  where names = nub $ map sname dep
```

```haskell
        getLine name = map toLine tracks
          where dtrack = track :: Departure -> String
                tracks = nub $ map dtrack $ filter ((== name) . sname) dep
                -- filter fusion, yay!
                getTrack tr = filter ((== name) . sname)
                                . filter ((== tr) . dtrack )
                concreteTime d = case rtTime d of Just t -> t; _ -> time d
                toLine tr = BL name tr $ map concreteTime $ getTrack tr dep

-- Req networking
```

A.4.3 Handling of Known Lines and the Main Loop

```haskell
{-# LANGUAGE TypeApplications, OverloadedStrings, CPP #-}
module Main where

import BusTimes
import InkyPhat

import Data.Maybe
import Control.Monad.Reader (lift)
import Data.Aeson
import Data.Time
import qualified Data.ByteString.Lazy.Char8 as B (pack)
import Data.Text hiding ( map, concatMap, zip
                        , length, replicate, any
                        , filter, null, take)
import qualified Data.Text as T
import Control.Concurrent
import Data.Text.Encoding as E
import System.Environment

import Data.List (sort)
import Control.Monad.Except


-- We have a flag in the cabal file that allows us to chose between using the
-- python interpreter for time and network queries or using a native haskell
-- implementation. However, the native implementation segfaults when run on
-- the device, due to a bug in the cross-compilation toolchain, hence the
-- possiblity to choose.
# if PYTHON_HACKS
import PythonHacks
# else
import ReqNetworking
# endif

getToken :: InkyIO (Maybe TokenResponse)
getDateTime :: InkyIO (Text, Text)
getBusTimes :: Token -> BusStop -> InkyIO (Maybe BusResponse)
# if PYTHON_HACKS
getToken = pyGetToken
getDateTime = pyGetDateTime
getBusTimes = pyGetBusTimes
#else
```

```haskell
getToken = liftIO hsGetToken
getDateTime = liftIO hsGetDateTime
getBusTimes t s = liftIO (hsGetBusTimes t s)
#endif


löviksVägen :: BusStop
löviksVägen = 9021014004663000
hovåsNedre :: BusStop
hovåsNedre = 9021014003235000

nearby :: [BusStop]
nearby = [löviksVägen, hovåsNedre]

setup :: InkyIO Image
setup = setRotation 180 >> image "vt" "vt.png"

fontheight :: Int
fontheight = 12
fontwidth = fontheight
linespace :: Int
linespace = 2


-- From Gabriel Gonzalez's tweet.
leftpad :: Char -> Text -> Int -> Text
leftpad c i n = T.replicate (n - T.length i) (singleton c) <> i

rightpad :: a -> [a] -> Int -> [a]
rightpad c i n = i ++ replicate (n - length i) c

printBusTimes :: [Text] -> InkyIO ()
printBusTimes msgs =
  do (inkyW, inkyH) <- dimensions
     let (x,y) = (2, inkyH `div` 2 - fontheight * length msgs `div` 2)
         pr (m,i) = text pos m Nothing Nothing
          where pos = (x, y + i*(fontheight + linespace))
     mapM_ pr $ zip msgs [0.. ]
  where pos = fromMaybe

printTime :: InkyIO ()
printTime = do (date, time) <- getDateTime
               (inkyW, _) <- dimensions
               let pos = ((inkyW - 16*fontwidth) `div` 2,18)
               text pos (T.unwords [date, time]) (Just Red) Nothing

printName :: InkyIO ()
printName = do (inkyW, _) <- dimensions
               let pos = ((inkyW - 14*fontwidth) `div` 2,2)
               -- Strætóferðir means "Bus Trips"
               text pos "Strætóferðir" (Just Red) (Just (Font 14))

printLogo :: Image -> InkyIO ()
printLogo img
 = do (inkyW, inkyH) <- dimensions
      (logoW, logoH) <- size img
      let (vx,vy) =  ((inkyW - 10*14 - logoW -10) `div` 2, inkyH -2-14)
          posText = (vx + logoW+2, vy -4)
          posLogo = (vx, vy-logoH+14-2)
      -- Västtrafik is the name of the Gothenburg transport authority.
      text posText "Västtrafik" (Just Red) (Just (Font 14))
      paste img posLogo

updateDisplay :: Image -> [Text] -> InkyIO ()
updateDisplay img times
 = do { clear
      ; printName
      ; printTime
      ; printBusTimes times
      ; printLogo img
      ; display }

-- Note [Known Lines]
-- We only want to show known lines, i.e. those that we take from our house,
-- and we want to display 82 first, then Rosa, and then the others.
-- Additionally, we want to show the 158 when it is running (in the morning),
-- but otherwise show the 82 going to Brottkärr, since we can only fit 3
-- lines comfortably on the display at once. This is why we do `take 3`
-- when choosing what to output, which selects the 158 if availble, but
-- the 82B otherwise.
data KnownLine = B82 [String]
               | BRosa [String]
               | B158 [String]
               | B82B [String] deriving (Eq, Ord)

-- These are the lines going in the right direction that we are interested in.
toKnownLine :: BusLine -> Maybe KnownLine
toKnownLine (BL "82" "A" d) = Just $ B82 d
toKnownLine (BL "82" "B" d) = Just $ B82B d
toKnownLine (BL "ROSA" "A" d) = Just $ BRosa d
toKnownLine (BL "158" "A" d) = Just $ B158 d
toKnownLine (BL n t d) = Nothing

knownDepartures :: KnownLine -> [String]
knownDepartures (B82 d) = d
knownDepartures (B82B d) = d
knownDepartures (BRosa d) = d
knownDepartures (B158 d) = d

instance Show KnownLine where
  show (B82 _) = "82"
  show (B82B _) = "82B"
  show (BRosa _) = "ROSA"
  show (B158 _) = "158"

renderLine :: TimeOfDay -> KnownLine -> Text
renderLine now kl =
   T.unwords $ map to5 $ [pack (show kl) <> ":"] <> dpts
 where to5 s = leftpad ' ' s 5
       dpts = rightpad "-" (map toDp $ knownDepartures kl) 2
       -- We want to show how many minutes there are until the bus departs.
       -- Note that we check for > 0, since the date might be in the past
```

```haskell
       -- if the bus is waiting at the stop or too much time has passed
       -- since the request was made.
       toDp t = if mins > 0 then pack (show mins) <> "min"
                else "Núna!" -- "Núna" means "Right now".
         where mins = floor (lt - nowSecs) `div` 60
               lt = timeOfDayToTime $ read @TimeOfDay (t <> ":00")
               nowSecs = timeOfDayToTime now

loop :: Image -> InkyIO ()
loop img =
  do (date, time) <- getDateTime
     -- We could save the token and re-use it if it hasn't expired, but since
     -- the performance is limited by the screen refresh time anyway, we don't
     -- bother.
     mb_token <- fmap access_token <$> getToken
     case mb_token of
       Just token ->
         do stops <- catMaybes <$> mapM (getBusTimes token) nearby
            let now = read @TimeOfDay ( unpack time ++ ":00")
                -- See Note [Known Lines]
                toDisplay = take 3 $ sort $ mapMaybe toKnownLine
                              $ concatMap getLines stops
                busTimes = map (renderLine now) toDisplay
                -- "Engar ferðir núna" means "No trips right now", which we
                -- display when the buses have stopped running (usually quite
                -- late at night).
                msg = if null busTimes then ["Engar ferðir núna!"] else busTimes
            updateDisplay img msg
       _ -> return ()
     liftIO $ threadDelay $ 60 * 1000 * 1000
     loop img

main :: IO ()
main = runInkyLoop $ setup >>= loop
```

A.4.4 Networking

```haskell
{-# LANGUAGE OverloadedStrings  #-}

module PythonHacks (
 urlRequest, pyGetToken, pyGetBusTimes, pyGetDateTime
) where

import Data.ByteString (ByteString)
import qualified Data.ByteString as B
import qualified Data.Text.Encoding as E

import qualified Data.Text as T
import Data.Text hiding (map, null)

import Prelude hiding (hPutStr, hPutStrLn, hGetLine, putStrLn)
import qualified Data.Aeson as JSON

import Control.Monad.Reader
import Control.Monad.Except
import BusTimes
import InkyPhat

-- The Nice request libraries segfault on the pi,
-- so we'll just use the already available python interpreter
data AuthHeader = Basic String | Bearer String

instance Show AuthHeader where
  show (Basic s) = show ("Basic " <> s)
  show (Bearer s) = show ("Bearer " <> s)


urlRequest :: Text -> [(Text,Text)] ->
              AuthHeader -> ByteString -> InkyIO Text
urlRequest url params auth body
 = readString readCmd
 where reqCmd = "request.Request(" <> intercalate "," args <> ")"
       readCmd = "request.urlopen(" <> reqCmd <> ").read().decode('utf8')"
       args = [ (pack $ show urlArg)
              , "headers={'Authorization':" <> (pack $ show auth) <> "}"]
              <> (if B.null body then mempty else ["data=b'" <> E.decodeUtf8 body <> "'"])
       urlArg = if null params then url
                else (url <> "?"<> ( intercalate "&" $ map paramToUrl params))
       paramToUrl (name,arg) = name <> "=" <> arg


-- | We use these to get the information via Python,
-- since the network libraries segfault on the pi
-- unless the TZ variable is set to a non-file base timezone.
-- Use these if you must.
pyGetToken :: InkyIO (Maybe TokenResponse)
pyGetToken = do (key, secret) <- liftIO getAuthCredentials
                let auth = authToken key secret
                    url = "https://api.vasttrafik.se/token"
                    body = "grant_type=client_credentials"
                JSON.decodeStrict . E.encodeUtf8 <$> urlRequest url [] (Basic auth) body

pyGetBusTimes :: Token -> BusStop -> InkyIO (Maybe BusResponse)
pyGetBusTimes token stop =
   do (date, time) <- pyGetDateTime
      res <- E.encodeUtf8 <$> urlRequest url (params date time) (Bearer token)  ""
      case JSON.eitherDecodeStrict res of
        Left err -> throwError $ ValueParseError (pack err)
        Right v -> return v
  where url = "https://api.vasttrafik.se/bin/rest.exe/v2/departureBoard"
        auth = Bearer token
        params date time = [ ("id", pack $ show stop)
                           , ("maxDeparturesPerLine", pack $ show 2)
                           , ("format","json")
                           , ("timeSpan", pack $ show 59)
                           , ("date", date)
                           , ("time", time)]


pyGetDateTime :: InkyIO (Text, Text)
pyGetDateTime = do d <- readString "datetime.now().strftime('%Y-%m-%d')"
                   t <- readString "datetime.now().strftime('%H:%M')"
                   return (d,t)
{-# LANGUAGE OverloadedStrings, TypeApplications #-}
module ReqNetworking (
  hsGetToken, hsGetDateTime, hsGetBusTimes
) where

import BusTimes

-- Req networking
import Network.HTTP.Req
import Data.Default
import Control.Monad.IO.Class
import Data.Time
import Data.Time.Format.ISO8601
import Data.Text (Text)
import qualified Data.Text as T
import Control.Retry

import qualified Data.ByteString.Char8 as B
import qualified Data.Aeson as JSON

config :: HttpConfig
config = def {httpConfigRetryPolicy = delay }
  where
    -- We default to delaying for 5 sec and trying 10 times.
    delay :: RetryPolicy
    delay = constantDelay (5000*1000) <> limitRetries 5

hsGetToken :: IO (Maybe TokenResponse)
hsGetToken = fmap Just $ runReq config $
  do (key, secret) <- liftIO getAuthCredentials
     let auth = basicAuth (B.pack key) (B.pack secret)
         url = (/~) @[Char] (https "api.vasttrafik.se") "token"
         -- We need a body, but how? Let's ask!
         body :: ReqBodyBs
         -- body = _ ("grant_type=client_credentials" :: ByteString)
         body = ReqBodyBs "grant_type=client_credentials"
     responseBody <$> req POST url body jsonResponse auth

hsGetDateTime :: IO (Text, Text)
hsGetDateTime =
 do now <- zonedTimeToLocalTime <$> getZonedTime
    return (toDate now, toTime now)
  where toDate = T.pack . iso8601Show . localDay
        toTime = T.pack . Prelude.take 5 . iso8601Show . localTimeOfDay

hsGetBusTimes :: Token -> BusStop -> IO (Maybe BusResponse)
hsGetBusTimes token stop = runReq config $
  do (date, time) <- liftIO hsGetDateTime
     res <- responseBody <$> req GET url NoReqBody bsResponse (auth <> params date time)
     case JSON.eitherDecodeStrict res of
        Left err -> liftIO $ print err >> return Nothing
        Right v -> return v
  where url = https "api.vasttrafik.se" /: "bin" /: "rest.exe" /: "v2" /: "departureBoard"
```

```
auth = oAuth2Bearer $ B.pack token
params date time = "id" =: stop
                <> "maxDeparturesPerLine" =: (2 :: Integer)
                <> "format" =: ("json" :: String)
                <> "timeSpan" =: (59 :: Integer)
                <>  "date" =: date
                <>  "time" =: time
```