Natural Program Synthesis from Examples in Haskell

Abstract
We present a new programming-by-example technique that efficiently synthesizes natural, readable fitting functions that combine user-defined higher-order functions with standard and third-party library code.

The search works by dismantling higher-order functions in order to deduce suitable refinement types. These refinement types are then used to prune the search space of possible higher-order functions for a given example set. Since refinement type under-approximate, we can apply Liquid Haskell to arbitrary syntax extensions while still preserving soundness.

We evaluate an implementation of our tool against a large set of synthesis examples including lists, trees, maps, and specialized musical score data structures. This evaluation demonstrates the scalability and versatility of this approach.

1. Introduction
Program synthesis is an active research direction [6, 14, 22, 25, 31, 32] that aims to automatically derive code from a given specification. This code is correct by construction and ideally would make a programmer more productive. Still, writing a complete specification of an entire program is often a more complex task than writing the corresponding code, even for very simple programs.

Programming by example [8, 13, 23] is a promising research direction that enables easy manipulation of data even for non-programmers [15]. Recent work in this area has focused on manipulating fundamental data types such as strings [12, 26, 30], lists [10, 27] and numbers [29]. The success and impact of this line of work can be estimated from the fact that some of this technology ships as part of the popular FlashFill feature in Excel 2013 [12].

Instead of writing code, the user provides a list of relevant input/output examples and the synthesis tool automatically generates a program that fits. In this way, the examples can be seen as an easily readable and understandable specification. In [11], the close connection between refinement types and examples is expounded through the lens of proof theory. Other works have explored further ramifications of the theory behind programming by example [13, 15, 27]. These theoretical foundations give us the power and direction to begin to make programming by example a mainstream feature of fully featured languages.

In order for programming by example to be useful in the context of a real language, synthesis cannot act as a closed system. Just as with code a user writes, the ability to reuse and edit synthesized code is an integral part of the programming process. Program synthesis does not inherently address the problem of code readability, often resulting in tools that produce something closer to an executable than the simple and stylistic code a human might write. Thus, our goal is to synthesize snippets that can be naturally integrated into code written by a programmer. In this paper we introduce an approach called natural synthesis that aspires to synthesize code that is not only correct, but that is natural and idiomatic to the language.

Though there is no formal definition of natural synthesis, we take it to mean the task of synthesizing programs from a specification, with the explicit purpose of finding programs that are both correct to the specification and comprehensible. The ideal natural synthesis procedure could pass something of a Turing test for writing code. A human inspecting the generated code should be able to understand it as easily as if it were written by a human.

Take the simple task of synthesizing a list flattening function from examples. Synthesis approaches that use only the primitive recursive operators [10, 27], would find a function similar to solution1 in Listing 1. Our tool instead focuses on actually utilizing common library functions. The results of this approach can be seen just below in solution2. In fact, solution1 is an application of the GHC[1] implementation of concatMap. To synthesize this solution using only the core higher order functions is certainly motivating, how-
ever a user would likely prefer to see a higher order-function `concatMap`, which exists in a standard library, if using synthesis with the goal of writing their own code. A natural synthesis system makes use of user defined functions and simulates program structures that commonly occur in the language. We choose to focus on functional programming, where a core part of the experience is writing higher order functions.

Functional programming encourages specifying general behaviors in the form of abstract, higher-order functions, and then filling in details with first-order functions later. In fact, many users write higher order functions first, then combine them in interesting and useful ways [24]. Library authors often provide users with a number of higher order functions to enable programmers to more easily write their applications. Since users write higher order functions with a deep understanding of the domain, using them in synthesis produces code that is more idiomatic and easier to understand than using generic higher order functions.

In order to facilitate synthesis over generic higher order functions, we run the synthesis algorithm in two stages. The first offline preprocessing stage infers rules about how the user’s higher order functions behave over their input and output types. We encode these rules using refinement types in Haskell with the LiquidHaskell tool [34]. Refinement types allow us to specify a stronger type signature by adding predicates about various properties of the types. In this work we only utilize the ability of refinement types to make judgments on the sizes of the inputs and outputs, to be explained in more detail in Section 5.1. These refinement types can then be utilized during the online synthesis stage, along with various type matching, ranking, and unification algorithms to efficiently prune and navigate the search space of solution programs. We show in the evaluation section that only a small number of examples is needed to synthesize clear and concise solution programs.

Although programming by example is an easy entry point for novice users, one of the drawbacks can be the tedious nature of the specification. For a user, writing out a sufficient number of examples for the synthesis tool to find a solution may involve specifying seemingly obvious examples such as `[] -> []` in order to cover base cases of recursion. However, much of this domain specific knowledge is encoded by the user defined functions, data types and library imports. By focusing our synthesis procedures on this space, we can reduce the number of required edge case examples and allow users to focus on the more natural examples.

In Listing 1, we have already seen the potential to size novel solutions to programming by example queries. However, a synthesis engine should also be able to synthesize novel (and sometimes unexpected) solutions to problems. Since the stated goal is to find simple programs that a human might write, this raises the question if finding “natural” and and novel programs are at odds with each other.

Our evaluation section showed that this is not the case. For example, natural synthesis of the Boolean “or” function finds two function, any (id) and foldr1 (max). The any (id) solution would expected by Haskell programmer, where any :: (a -> Bool) -> [a] -> Bool is a built-in function to Haskell that returns True if any element of a list satisfies the predicate function. The more novel solution returned by our natural synthesis is foldr1 (max), where max :: Ord a => a -> a -> a will return the maximum element of the two inputs. By folding over the list, this solution program exploits the `Enum` property of the Boolean type in Haskell in a way that provides insight into some core Haskell functions and types.

In summary, we present the following contributions:

1. A programming by example system for a real language (Haskell) that uses natural synthesis to generate simple, reusable code.
2. Uniform handling of user-defined data types, first-order functions, and higher-order functions, as well as third-party libraries.
3. A procedure to automatically find suitable refinement types for user-defined higher order functions to prune the search space for our synthesis procedure. Multiple passes of the type-match-ranking algorithm sort, shrink, and – when needed – expand the search space.
4. An evaluation of the performance of our tool, as well as examples of the code that it can synthesize. The benchmarks presented show our tool can efficiently generate a wide variety of code that mixes functions from multiple sources.

## 2. Motivating Examples

### 2.1 Synthesis with the standard library

As an introduction to our tool, imagine that a user wants to synthesize the simple `stutter` function which duplicates each element of a list. While in MYTH [27] the focus was on lists as inductively defined data types, we are focusing on using the built in representation of a list in Haskell.

Haskell provides two relevant functions for this task in the standard library; `concatMap :: (a -> [b]) -> [a] -> [b]` which applies a function over a list and concatenates the result, and `replicate n :: Int -> a -> [a]` which will replicate an item n times into a list. When working over the inductively defined
list data type, MYTH requires three examples to find an appropriate function. Using our tool, the user only needs to provide a single example to synthesize the program \texttt{concatMap ( replicate 2)} that fits that example on Haskell’s built-in list type.

\begin{verbatim}
1 exs :: [(Int) -> [Int]]
2 exs = [[1, 2, 3] -> [1, 1, 2, 2, 3, 3]]
\end{verbatim}

If, in addition, the user develops a function \texttt{dupl} which duplicates an element (see below), then our tool will provide the solution \texttt{concatMap dupl}, as well as the solution listed above. Our tool will rank solution programs that contains user defined functions above others, so this one will be reported earlier. Ranks are calculated throughout the algorithm, continually being updated as we derive new information.

\begin{verbatim}
1 dupl :: a -> [a]
2 dupl x = [x, x]
\end{verbatim}

In the next example the user would like to synthesize a function that takes numbers from a list as long as the numbers are odd. Again, only a single example is needed for our tool to unambiguously find simple solution program, \texttt{takeWhile odd}, using functions the from the standard library; \texttt{takeWhile :: (a -> Bool) -> [a] -> [a]} to recurse over a list and take items until the predicate is false and \texttt{odd :: Int -> Bool}.

\begin{verbatim}
1 exs :: [(Int) -> [Int]]
2 exs = [[1, 2, 3] -> [1]]
\end{verbatim}

Another correct solution for this example might be \texttt{head} to take the first element. Searching for first order functions is an active research direction \cite{12, 20}, but in this work we are focused only on higher order functions. This is part because the goal of natural synthesis is to provide useful, nontrivial functions to users. A discussion of integrating our tool with first order searching techniques is provided in Section 7.

2.2 Synthesis with user defined values

Working on a set of user defined code is also a critical task our tool supports. In the next example the user has provided a binary tree data structure and a higher order function to map over it. We show the synthesis of the exceedingly (for the sake of brevity) simple program \texttt{mapBTree not}. Doing such synthesis requires automatic reasoning about not only the user defined polymorphic data type, but also the higher order function they have defined over it.

\begin{verbatim}
1 data BTree a = Nil | Branch (BTree a) a (BTree a)
2 mapBTree :: (a -> b) -> BTree a -> BTree b
3 mapBTree f Nil = Nil
4 mapBTree f (Branch bl v b2) = Branch (mapBTree f bl) (f v) (mapBTree f b2)
5 exs :: [BTree Bool -> BTree Bool]
6 exs = [Branch Nil True Nil ->]
7 exs = [Branch Nil False Nil ->]
\end{verbatim}

It may seem that if a user can write a higher order functions over custom data structures, they would not have a need to synthesize such functions. However, imagine the case of a user importing libraries. Haskell’s module system and large repository of libraries like Hackage and Stackage are indispensable part of the language\cite{2, 5}. Often, a user is importing a library that is large, unfamiliar, and/or poorly documented. Using our tool, the user no longer needs an intimate knowledge of the library to makes use of the functions and datatypes, and can instead synthesize functions from examples.

2.3 Synthesis with a DSL

As an example, we show code to transpose a music value from the Euterpea DSL (domain specific library) for music\cite{19}. Among other things, Euterpea defines a tree-like datatype called Music and various functions for manipulating these types. The user only needs to express the basic datatype as examples, and our tool can synthesize the solution program. The solution utilizes the functions from Euterpea; \texttt{mMap} for mapping over music values, and \texttt{(trans :: Int -> Music -> Music)} to transpose a Music Pitch by a value. This again requires automated reasoning about the properties of the library’s data types and higher order functions. Because we have synthesized a natural looking program, the user does not need to understand details of the library’s function and data structures to be able to immediately gain an intuition about how the solution program works.

\begin{verbatim}
1 import Euterpea
2 exs :: [Music Pitch -> Music Pitch]
3 exs = [ (Prim (Note qn (C,4)) :+: Prim (Note qn (D,4))) ->
4       (Prim (Note qn (D,4)) :+: Prim (Note qn (E,4))) ]
5 solution = mMap (trans 2)
\end{verbatim}

3. Problem Formulation

Synthesizing correct programs is a well researched problem; however, if programming-by-example is to become a mainstream tool for programmers, the synthesized code must be easy for a human to read and modify. The aim of our tool is to synthesize programs from examples that utilize user defined code in a clear and concise. We focus particularly on data structure manipulation problems that can be solved with higher order functions.

3.1 Example Syntax

Formally speaking, an \textit{example} is a pair of values with distinguished “input” and “output” elements, and an \textit{example set} is a set of examples all of whose inputs are of like type, and all of whose outputs are of like type. The output type does not necessarily match the input type.
A user supplies examples via a custom pair constructor \( \rightarrow \). This operator is used to differentiate between generic pairs and examples, but does not confer any additional structure. We require all higher order functions to be of a unified signature \( \rightarrow \ast \rightarrow \ast \), where the final kind of the signature is a function mapping the input type to the output type. Here, a kind is understood to be the type of a type constructor, in this case \( \rightarrow \), which constructs a function type from two other types.

The practical consequence of this format is that a user must partially uncurry (collapsing trailing function arguments into a single tuple argument) any higher-order function they are interested in using during synthesis. This also means that any type variable appearing in the higher-order function must be accounted for in the input and output types so that all type variables in its signature can be resolved. This allows us to conclude that any types that are between the input and first order function will be static initial values, which can be assigned using the process described in Section 6.3. This is a simple procedure that makes use of the user’s domain knowledge of which parameters to the function will be given by the examples; consider:

1. \( \text{zipWith' :: } (a \rightarrow b \rightarrow c) \rightarrow ([a], [b]) \rightarrow [c] \)
2. \( \text{zipWith' } f (xs, ys) = \text{zipWith } f \text{ xs ys} \)

### 3.2 Solution Space

By formally defining the space of functions we are interested in synthesizing, we can this definition to prove some properties on the algorithm. In particular we show in Section 7.1 that our tool is complete for this subset of functions.

The solutions our tool supports synthesizing are higher-order data structure manipulation programs. The higher-order functions take a component function that is a first-order function, for example \((+)\). The solution programs can be expressed as:

1. \( \text{solution :: } \)
2. \( \rightarrow \ast \rightarrow \) types -- Component Function
3. \( \rightarrow \) types -- Initial Values
4. \( \rightarrow \ast \) -- Input
5. \( \rightarrow \ast \) -- Output
6. \( \text{types } = \ast | \ast \rightarrow \) types
7. \(-- \ast \) matches on type variables and constructors.

Generally, the component function is applied across the input data structure, which the solution uses to construct an output data structure or reduction. As we will argue in Section 7 this set is expressive enough to support the classic \text{map}, \text{filter}, and \text{fold} functions, as well as higher order functions found in imported modules and user-supplied code.

Our goal is to create a synthesis procedure that is easily portable across full implementations of functional languages (Haskell, OCaml, etc), so we prefer using a type directed approach to synthesis over explicit code analysis whenever possible. This increases the portability and longevity of our system. For this implementation we target Haskell, detailing the exact modifications needed to expand this to other languages in Section 7.3.

### 4. System Overview

Figure 1 gives a high-level description of ways in which the components of our algorithm interact. Broadly speaking, there are two main stages in the algorithm. The offline (preprocessing) phase gathers the higher order declarations visible in the APIs and user-provided code, and assigns refinement types to them to build a custom synthesis engine. This engine is then used during the online phase of the algorithm to search for functions that fit a set of supplied examples.

During the offline phase, the algorithm first scans the user-provided code, the libraries it imports, and the standard library to gather all of the functions and global values visible to the program. Then, it selects the higher-order functions from the set of all functions and values, and uses Liquid Haskell [33–35] to assign refinement types to them to build a custom synthesis engine. This engine is then used during the online phase of the algorithm to search for functions that fit a set of supplied examples.

Once this stage is complete, the user can examples to the synthesis engine, which will search the space of constructable functions for those that fit the examples. First, the engine computes a refinement type that fits the examples. This type is matched against the refinement types of the known higher-order functions, and the weights of each
known function are adjusted based on how close the types match, if at all.

Once the candidate higher order functions have been chosen, the synthesis engine performs a best-first search for a program that fits all of the input and output examples by composing the candidates with first-order functions. For example, the higher-order function `map` might be supplied the `length` function if the example inputs are lists of lists of integers and the output examples are all lists of integers. The programs that are examined during the search are evaluated against the example set and are reported to the user as they match. Because the weights favor local declarations, the highest-ranked programs are likely to be the most idiomatic.

We present a pseudocode algorithm here which we will use as a roadmap for the rest of the paper, explaining each line in the proceeding sections. For the remainder of the paper, code samples are taken from the implementation, and modified slightly to elide the details of managing Haskell’s type system.

**Listing 2.** A pseudocode representation of the build and synthesis stages of the synthesis algorithm

```haskell
main = do
    eng ← build
    ex ← getExamples
    synth eng ex

    build = do
        allTypes ← collectTypesAndWeights
        allHOTypes ← filter isHigherOrder allTypes
        allRTypes ← assignRTypes allHOTypes
        return (allTypes, allRTypes)

    synth eng ex = do
        exType ← getExampleType ex
        exRType ← assignRTypes exType
        -- make candidate functions and programs
        hoFxns ← rankByTypeMatch exRType eng
        progs ← makeFxns exType hoFxns
        -- test the ranked list of possible programs
        validProgs ← filter (testOn ex) progs
```

### 5. Offline: Synthesis Engine Construction

#### 5.1 Refinement types for higher order functions

The first step of our algorithm (line 7 of Listing 2) is to collect all of the type signatures from our sources (user code, imports, and standard library). In order to rank the higher-order functions, we assign weights based on their source location. User-defined functions are given the highest priority, while direct imports are given less, and the standard libraries are given the least. These rankings will contribute to the final ranking of candidate functions in the synthesis stage when we match the component function signatures on the examples.

We filter through these to select only the higher order functions. Because in Haskell the function type constructor (→) is right binding, any higher order functions must have parenthesis in the type signature, which provides a convenient filtering predicate. This is over approximating filter, since the type signature might contain extraneous parentheses, for example surrounding the entire signature. In practice, it is rarely the case that a programmer will add extraneous parentheses to type signatures and this does not significantly impact performance.

In line 9 of Listing 2 we call the `assignRTypes` function (shown in Listing 3) to automatically generate refinement types that relate input and output sizes for our higher order functions. These refinement will be used to prune the search space in synthesis, as explained in detail in Section 6. In brief, the size relation that applies to the higher order functions must also apply to the examples in order to consider that function as a candidate. When writing the refinement types, we can be sure, by our specification in Section 3, that the last two types are always the input and output. We can then make use of type holes, in order to account for the diversity of component functions and initial values that might be required for any given higher order function.

```haskell
map :: i : [ a ] → { o : [ b ] | ( len i ) = ( len o ) }
```

For every predicate we test against, we are able to more accurately prune the search space of higher-order functions. However, since we must test many higher-order functions on each these predicates, the cost to add a predicate is high. Therefore, it is best to only select as many refinement types as are needed. We only use predicates of ≤, =, ≥ to specify size constraints on input and output. Notice that map will actually satisfy all three of these predicates, which in general, results in an over approximation of appropriate refinement types for higher order functions. We discuss potential optimizations for this in Section 7.

**Listing 3.** Adding refinement types to higher order functions

```haskell
assignRTypes :: Sig → IO (Sig, [ RType ])
assignRTypes sig = do
    x ← if eqTypes (lastTypes sig)
        then rTypeAssign sig
        else return [ noRType ]
    return (t, x)

    testRs :: Sig → IO ([ RType ])
    testRs s =
        filterM (runLiquidHaskell s) allPossibleRTypes
```

We separate possible types into two cases using the `eqTypes` function on line 3 of Listing 3. In the case that input and output types (extracted with `lastTypes` `sig`) of the higher order functions are the same (up to equality on the top level type constructor), we should generate refinement types. In the other case, when the input and output type are different, the size measures between two different type constructors are not guaranteed to have any significance. A relation on these values may be useful on occasion, but in practice is more often only a confounding factor, leading to wasted
computation. When we do not assign refinement types to a higher order function, we tag the higher order function with the placeholder `noRType` value. These `noRType` tagged functions can be further pruned in the synthesis stage by utilizing a subtype ranking system to be explained in more detail in Section 6.

5.2 User defined data types

In order to support user defined data structures, we only require that a user implements some kind of measure\([34]\) over their data structure. This size function will allow LiquidHaskell to determine size constraints on the examples, so that our tool can pick higher order functions that also satisfy those size predicates. In fact, the size function could just be a constant function, resulting in every function and example satisfying the equality refinement type predicate. This means the system will test every higher-order function that fits the types.

As an example, take the code from Section 2 for synthesizing a music function. The user would have needed to provide a measure function for Music. a. This measure will allow LiquidHaskell to draw conclusions about the size of examples of type `{Music a : \rightarrow \text{Music a}}$, as well as conclusions about higher order functions over the Music data structure. In this context, one sensible measure function counts the number of notes (Prim) in the tree-like Music structure, as shown by the `len` function in Listing 4.

<table>
<thead>
<tr>
<th>Listing 4.</th>
<th>a user defined measure over a datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>import Euterpe</td>
<td></td>
</tr>
<tr>
<td>{ measure \text{len} @-}</td>
<td></td>
</tr>
<tr>
<td>\text{len} :: \text{Music a} \rightarrow \text{Int}</td>
<td></td>
</tr>
<tr>
<td>\text{len} \text{m} =</td>
<td></td>
</tr>
<tr>
<td>\text{case} \text{m} of</td>
<td></td>
</tr>
<tr>
<td>Prim _ \rightarrow \text{1}</td>
<td></td>
</tr>
<tr>
<td>\text{ml} :+\text{m2} \rightarrow \text{len} \text{ml} + \text{len} \text{m2}</td>
<td></td>
</tr>
<tr>
<td>\text{ml} :\sim \text{m} \rightarrow \text{len} \text{ml} + \text{len} \text{m2}</td>
<td></td>
</tr>
<tr>
<td>Modify \text{c m} \rightarrow \text{len} \text{m}</td>
<td></td>
</tr>
</tbody>
</table>

6. Online: Fitting Functions to Examples

With the synthesis engine constructed, the system is ready to synthesize programs from examples. Multiple programming-by-example queries can then be answered using this engine. The synthesis engine only needs to be reconstructed when there are new library imports, or when there is a revision of the user-supplied code.

When examples are provided, the synthesis engine finds a suitable refinement type for a hypothetical function that could fit that example. Then, our tool filters and ranks the higher order functions based on the refinement types known to the engine and the example types provided. Once the candidate higher functions are identified, our tool will select and build first order functions that match the type of the higher order function’s component signature to build a final set of candidate programs.

Each of these candidate programs is executed in best-first order against the set of inputs. Whenever a function produces the correct outputs for each input, it is said to fit, and is reported to the user. This search continues until the space is exhausted or it is manually interrupted. The search will always terminate since we are working over a finite space of generated functions, are our type reductions are strictly decreasing, which we will explain in Section 6.2.

6.1 Refinement types for examples

As in Section 5.1, we also consider two cases for examples. The first, where the example input and output types match up to the top level type constructor, and the second where the types do not match.

In the case that the types do match, we find the set of refinement types that the examples satisfy. Generating refinement type predicates about the size of the input and output, as in Section 5.1, we apply the same algorithm from Listing 3. For instance, an example set for `filter (>3)` might look as follows:

<table>
<thead>
<tr>
<th>Listing 5.</th>
<th>Refinement type inference for examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ex :: {\text{Int}} \rightarrow {\text{Int}}</td>
<td></td>
</tr>
<tr>
<td>\text{ex} = {1, 2, 3} \rightarrow {1, 2, 3}</td>
<td></td>
</tr>
<tr>
<td>{1, 3, 4} \rightarrow {1, 3}</td>
<td></td>
</tr>
<tr>
<td>{4, 6, 8} \rightarrow {\text{1}}</td>
<td></td>
</tr>
<tr>
<td>exRTyp ::</td>
<td></td>
</tr>
<tr>
<td>\text{ex} \text{RTyp} ::</td>
<td></td>
</tr>
<tr>
<td>\text{inExs} :: {\text{Int}}</td>
<td></td>
</tr>
<tr>
<td>\rightarrow</td>
<td></td>
</tr>
<tr>
<td>{\text{outExs} :: {\text{Int}}</td>
<td>\text{len} \text{inExs} \leq \text{len} \text{outExs} }</td>
</tr>
</tbody>
</table>

and have the final refinement type of `exRTyp`, since all of the examples suggest that the output list does not grow. Again, when the types do not match we assign the `noRType` flag to the examples, as we did for higher order functions in Listing 3. We can now reduce our search space to only higher order functions with the same refinement type that matches the examples’ refinement type.

6.2 Type match ranking

Once our tool has both the base and refinement types for the examples and higher order functions, it can can prune and order this set (line 17 of Listing 2). The first step is to simply filter the higher order function candidates over equality of refinement types. Additionally, our tool will check the example types are concrete versions of the input/output types of the higher order function with the infix (for clarity) isConcreteTypeOf function. For type `A` to be a concrete version of type `B`, there must exist some type `C` (possibly equal to type `B`), such that both `A` and `B` can be instantiated to that type. The above requirement is then that there is some way to unify these two types - a familiar problem\([7]\).

<table>
<thead>
<tr>
<th>Listing 6.</th>
<th>Pruning based on types</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{filter} (\text{exRTyp} ==) \text{higherOrderRTypes}</td>
<td></td>
</tr>
<tr>
<td>\text{filter} (\text{exRTyp } \text{isConcreteTypeOf}) \text{higherOrderComponentTypes}</td>
<td></td>
</tr>
</tbody>
</table>
Once these higher order functions have been culled from the pool of candidates, we update their ranks that had been assigned in Section 5.1 from code locality. The higher order function can advance in the ranking by using a value function to find out exactly how much the example type isconcreteTypeOf to the input/output types of candidate higher order function.

In Listing 7, we present a demonstration of part of this ranking algorithm. As we traverse the tree structure of the type, the more pieces of the type signature that match, the higher the value of that match. However, if there is a type constructor mismatch, the two types can never be reconciled, and the entire value gets nothing.

Listing 7. Type closeness ranking algorithm (sample)
```
1 value :: Type -> Type -> Maybe Int
2 value (TyFun i1 o1) (TyFun i2 o2) =
3   1 + (value i1 i2) + (value o1 o2)
4 value (TyCon n1) (TyCon n2) =
5   if (n1==n2) then 20 else Nothing
6 value (TyCon n1) (TyVar _) = 10
7 value _ _ = Nothing
```

As an example of how this value function is applied the higher order functions, imagine we have three map functions specialized on particular values. The fully polymorphic map will score 1 point for having a function between input and out, 2 points for both having lists, and 20 points for a type variables matching a type constructor, for a total of 5 points. The mapI for Ints, will score the same, but score 20 points for a type constructor mismatch, the two types can never be reconciled, and the entire value gets nothing.

```
-- mapB scores Nothing
-- mapI scores 43
-- map scores 5
```

6.3 Component function generation

Recalling the solution space of programs defined in Section 3, our tool must now find first order function for each of the higher order functions that are still candidates (line 18 of Listing 2). For a given higher order function, our tool can choose component functions by reusing the weighted type matching algorithm from Listing 7. Since examples must be given as a concrete type, we can always partially specialize our candidate higher order function. We then search for first order functions that will type check against the partially specialized component signature. This partial specialization is a way of extracting more information out of our examples, and significantly reduces the space of candidate first order functions. Similar to Listing 8, we show an example of how type matching is applied over first order functions in Listing 9.

```
Listing 9. Ranking component function
1 examples :: [Int] -> [Int]
2 map :: (a -> b) -> [a] -> [b]
3 mapI :: (Int -> Int) -> [Int] -> [Int]
4 component :: Int -> Int
5 f1 :: a -> b  -- value is 21
6 f2 :: Int -> a  -- value is 31
7 f3 :: Int -> Int  -- value is 41
8 f4 :: [Bool] -> [Bool]  -- value is Nothing
```

6.4 Initial Values

In addition to finding first order functions where the arity of the kinds is equal to the component function, we may also want “larger” functions that have been applied to initial values. For examples, if the component signature is :: Int -> Int, we may choose the first order functions (+) :: Int->Int->Int in scope. By applying some initial values to (+), we can get a new function (e.g. (+1) :: Int->Int) that fits the component signature.

If the initial value’s type is an instance of Monoid, we can extract the unit value (named mempty in Haskell’s monoid typeclass[3]) to use as our initial value. For lists, the unit element is []. However, there are two valid monoids for numbers, using either (+) or (*) as the operators and resulting in unit elements 0 and 1 respectively. We take both of these values (along with other common, useful values of -1, and 2) as possibilities since the cost of testing both values is relatively small.

Additionally, requiring our users to write monoid instances for their datatypes may be a nuisance. However, users may have some domain knowledge that a particular value, or set of values, may be useful in their application. Since our system automatically considers functions defined in the user code base, users may simply write their own specializations of the higher order functions, or provide useful initial values, to be used in synthesis.

```
Listing 10. adding default initial values
1 -- to use 5 as an initial value for foldl
2 foldl :: (a -> b -> a) -> [b] -> a
3 foldl5 f i o = foldl f i o
4 -- to use 5 as an initial value in all recursions
5 x :: Int
6 x = 5
```

Presented with the problem of finding integer values to satisfy the examples may initially seem like a good application for an SMT solver. However, keep in mind that we do not in general know what we are trying to solve - the actual use of these variables is hidden within the function definition. Since in this work we maintain a primarily type
directed approach, rather than code analysis, we will not be able to unravel these functions.

We must also address the issue first presented in Section 3, that it is possible for a higher order function to need initial values in addition to a component function. For example, the map function only takes a first order function, while foldl \( \cdot \) \( \cdot \to \cdot \to a \to [b] \to a \) requires an initial value for a. Using a similar process as for first order function application, we can apply values until the higher order function only needs the example input to complete execution. To identify initial values in a higher order type signature, we can use our previous assumption that all higher order function have been partially curried to the type \( \cdot \to \cdot \to \cdot \). Adding the further assumption that only one first order function maybe be passed to the higher order function, we simply tag any non-function type in the hole as an initial value.

7. Evaluation

7.1 Soundness and Completeness

It is clear that no function will be returned by the algorithm that does not fit the examples given, since functions are validated before being reported. Therefore, it is trivial to conclude that our tool is sound over the given examples. Still, it is possible for the synthesis procedure to return a function that does not capture the user’s intent - that is, as with any programming by examples system, our tool is not sound over the user intent. Generally, this ambiguity can be resolved by the user supplying more examples to narrow the set of possible fitting functions. However, depending on what the user is trying to synthesize, and which examples have been provided, it is possible for new examples to increase the internal search space. If, for example, a user gives only positive examples for a filter, the refinement type predicate discovery will assume that the lists do not change size, and will likely return map id as a result.

The completeness claim we might like to make is that over the solution space defined in Section 3, we will always find a solution if it exists. Since our space is finite, completeness can be made trivially true by replacing all instances of pruning with a zero ranking, so that our algorithm now is only a best-first enumerative search. Because we make some decisions in pruning that removes potentially sound functions, such as using the noRType tag in Section 5.1 we trade this completeness for performance. In Section 8, we will discuss why, even if we had completeness, it should be sacrificed in future work.

7.2 Performance

In Table 1 we show detailed information about our tool over a set of benchmarks. These benchmarks were chosen to show the versatility of our tool over many different applications and libraries. The benchmarks over booleans, trees, and lists are common to many other programming-by-example tools. The examples that utilize the Data, List and Euterpea libraries to show our tool’s ability to work with large, highly specialized, 3rd-party libraries. Due to the algorithm’s focus on generating natural code, the synthesized functions are concise enough to be listed within the table itself. The representative examples show that few, simple hints to the synthesizer are able to produce good results. In many cases, the representative examples are actually the only examples necessary to synthesize the desired function. This shows that our approach uses the information available to it effectively.

In addition, the runtime average about ten seconds thanks to the inherently parallel nature of the search. With just a few lines of code, we were able to achieve order-of-magnitude speedups over the serial version. Haskell’s functional parallelism model is ideal for embarrassingly parallel problems like this one, and promises good scaling to larger instances of the problem over increasing computational resources.

In Section 5.1 we discuss how type matching and the noRType tag reduce the number of refinement type inferences we make. Recall that even if both types have a measure (lists and trees), in general there is no guarantee that this is a meaningful comparison. Since LiquidHaskell is the largest cost to our system in the offline stage, removing refinement type inference in these ambiguous cases provides a large performance gain. As an example, in processing the Haskell standard library base: Prelude, 7 out of 30 higher order functions do not need to be checked against refinement types using this approach.

7.3 Example Generation

We have tried to avoid code analysis at every stage of this paper. However there are two points where this has fallen short. First, we must parse a file to extract the name and type information of every top level identifier. Second, using LiquidHaskell as a blackbox means that we are limited by LiquidHaskell’s ability to deduce refinement types over functions. Our eventual goal is to create a system that can be easily ported across functional languages. Luckily, the first code dependency is small enough to handle with ease in most typed languages (the grammar of a type signature is relatively small). However LiquidHaskell is a powerful tool that would be difficult to recreate in another language.

To this end, we can extend the refinement type system by allowing refinement type inference on representative examples of a higher order function. We do not need to identify a particular component function since we are only interested in size based refinement types. We then apply a similar refinement type inference strategy as in Listing 5 to these examples.

Our current example generation tool uses QuickCheck to generate and apply many examples for higher order function Haskell [4]. Of course, since LiquidHaskell supports so much of Haskell, this is not practically useful for us, but provides a prototype as a proof of concept. Imagining that we could not find a refinement type directly on map, we might
<table>
<thead>
<tr>
<th>Name</th>
<th>Time (s)</th>
<th>Imports</th>
<th># Ex</th>
<th>Representative Example</th>
<th>Generated Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>2.02</td>
<td>None</td>
<td>3</td>
<td>[True, False] :→ False</td>
<td>all id</td>
</tr>
<tr>
<td>and-2</td>
<td>5.52</td>
<td>None</td>
<td>3</td>
<td>[True, False] :→ False</td>
<td>foldl min True</td>
</tr>
<tr>
<td>or</td>
<td>3.95</td>
<td>None</td>
<td>4</td>
<td>[True, False] :→ True</td>
<td>any id</td>
</tr>
<tr>
<td>xor</td>
<td>5.59</td>
<td>None</td>
<td>4</td>
<td>[True, False, True] :→ False</td>
<td>foldl xor False</td>
</tr>
<tr>
<td>double vals</td>
<td>3.35</td>
<td>None</td>
<td>1</td>
<td>((1) 3 (2)) :→ ((2) 6 (4))</td>
<td>mapBTree (*2)</td>
</tr>
<tr>
<td>tree id</td>
<td>2.49</td>
<td>None</td>
<td>1</td>
<td>((1) 3 (4) (5) (6)) :→ ((1) 3 ((4) (5) (6)))</td>
<td>mapBTree id</td>
</tr>
<tr>
<td>tree max</td>
<td>2.95</td>
<td>None</td>
<td>3</td>
<td>(1 10) 5 :→ 10</td>
<td>accumTree max 0</td>
</tr>
<tr>
<td>tree sum</td>
<td>2.93</td>
<td>None</td>
<td>1</td>
<td>((3 1) 2) :→ 6</td>
<td>accumTree (+) 0</td>
</tr>
<tr>
<td>all even</td>
<td>2.02</td>
<td>Data.List</td>
<td>4</td>
<td>[2,4,6,8] :→ True</td>
<td>all even</td>
</tr>
<tr>
<td>some odd</td>
<td>4.70</td>
<td>Data.List</td>
<td>3</td>
<td>[1,4,5,6] :→ True</td>
<td>any odd</td>
</tr>
<tr>
<td>custom filter</td>
<td>11.88</td>
<td>Data.List</td>
<td>3</td>
<td>[1,2,3,4,5] :→ [3,4,5]</td>
<td>filter_user_pred</td>
</tr>
<tr>
<td>length</td>
<td>1.20</td>
<td>Data.List</td>
<td>3</td>
<td>[5,6,7,8] :→ 4</td>
<td>foldl count 0</td>
</tr>
<tr>
<td>negate all</td>
<td>7.48</td>
<td>Data.List</td>
<td>1</td>
<td>[True, False, True] :→ [False, True, False]</td>
<td>foldl max 0</td>
</tr>
<tr>
<td>odd prefix</td>
<td>8.77</td>
<td>Data.List</td>
<td>1</td>
<td>[1,3,4,6,7] :→ [1,3]</td>
<td>map not</td>
</tr>
<tr>
<td>stutter</td>
<td>3.02</td>
<td>Data.List</td>
<td>1</td>
<td>[1,2,3] :→ [1,1,2,2,3,3]</td>
<td>takeWhile odd</td>
</tr>
<tr>
<td>sum ints</td>
<td>4.64</td>
<td>Data.List</td>
<td>1</td>
<td>[1,2,3,4] :→ 10</td>
<td>concatMap (replicate 2)</td>
</tr>
<tr>
<td>set sum</td>
<td>1.83</td>
<td>Data.Map</td>
<td>1</td>
<td>{ 1, 2, 3, 4 } :→ 10</td>
<td>Data.Map.foldl (+) 0</td>
</tr>
<tr>
<td>music id</td>
<td>7.47</td>
<td>Euterpea</td>
<td>1</td>
<td>C# :→ C#</td>
<td>mMap id</td>
</tr>
<tr>
<td>transpose score</td>
<td>5.15</td>
<td>Euterpea</td>
<td>1</td>
<td>A :→ B</td>
<td>mMap (trans 2)</td>
</tr>
</tbody>
</table>

Table 1. Benchmarks and Performance Measures. This table lists all 20 benchmarks, grouped by data structure. Each benchmark lists its name, the amount of time it took to synthesize, the extra imports it uses, the number of examples needed to synthesize, one representative example, and the synthesized function itself. The group marked “Tree (u.d.)” is a user-defined structure with user-defined higher-order operations. All reported data is generated on a Linux machine with four cores of Intel i5-3450 @ 3.10GHz and 8 Gb of ram.

use examples to infer a refinement type. Take the following code:

```
map     :: (a → b) → [a] → [b]
map f [] = []
map f x:xs = f x : map f xs
mapExs  = [[1, 2, 3, 4] :→ [4, 2, 8]]
```

There are however repercussions to this approach. We are not guaranteed to generate a correct refinement type because we might not generate a fully representative examples. It then seems it is possible prune away many high order functions that are actually useful, but the full repercussions of this is outside the scope of this paper.

8. Related Work

In describing our tool, we have shown how relying on a type-directed synthesis approach frees us from burdensome constraints of code analysis and hard coded inference rules, and allows our tool to synthesis natural and organic code. Many of the techniques we have used have been explored in various contexts before, though generally for the purpose of lower level synthesis. In this section we make some comparisons to related work, and highlight the differences we employ that help us generate readable code.

One of the most closely related works in aspirations is MagicHaskeller [20]. This project makes heavy use of a ranking system based on code use and lookup frequency in a database to deliver natural results to the user. In contrast to our work, MagicHaskeller uses a database of functions as its main synthesis engine, with the current database hovering around 64GB [21]. From this work, we take the inspiration of supporting imported libraries for creating natural code. However, it is important that the system is more portable and easily manipulated by the user - in particular by allowing user defined function in synthesis.

MagicHaskeller work is in the same AI focused domain of inductive programming as the tool IgorII [18]. IgorII however takes a very code analysis heavy approach, having been originally developed for Maude, then ported to Haskell.

One of the motivating works for exploring type-directed programming by example, especially over recursively defined datatypes is MYTH [11, 27]. The natural extension of this work in the usability direction was to include a more lightweight and flexible support for user defined and imported datatypes. The A^2 tool also focuses on deriving programs over recursively defined datatypes [9]. One of the major barriers to an average user with these tools, is that the generated code operates on the inner workings on a datatype.
While this provides a complete picture of all the data manipulation, often a user might prefer to simply be provided with high level, functioning code. Building in support and the ability to reason on source defined functions in our tool has made this natural synthesis possible.

Another feature these works support is the synthesis of first order function. According to our problem specification in Section 3, we are only focused on synthesizing higher order functions. One common synthesis problem these works present is to append an item to a list. If we were to extend our strategy to first order functions, our tool would just search through the libraries to find the append function, as the most natural solution. With the eventual goal of building a complete program synthesis engine, we will need to integrate with more advanced first order function synthesis systems. While this problem has been investigated in isolation, it is not clear how to efficiently determine if a set of examples more naturally calls for a higher order function or a first order function.

One direction to explore for first order synthesis is the type reduction algorithm in Section 6.4. In providing initial values for component functions, we current are strictly reducing the number of kinds in a type signature. While this gives us a termination of search guarantee and completeness over the search space, that is not particularly useful guarantee in this case. Imagining our tool integrated with an IDE, it would be better to keep the tool running constantly to infinitely search for new suggestions to the code. To this end we could also enable non-reducing types reducing applications, which would create an infinite recursion, but allow us to find many more functions. Rather than supplying values to functions, we could supply more functions.

```
-- given a component function
f :: Int → Int
f = (+1)

-- apply a function rather than a value
f' :: Int → Int
f' = f . (+2)
```

The relationship between refinement types and examples is explored in detail in [11], leading them to use refinement types to extend the specification language of a programming by example system. Instead, our tool keeps refinement types entirely as a backend logical inference technique and hidden from the user. In line with our goal of synthesizing natural code, we wish to minimize the asking the onerous task of users to learn to write new specifications. Simple as they may be, refinement types are unlikely to seem as approachable as the more familiar “examples as a specification” to the average user.

Taking the refinement types as a specification even further, [28] proposes a system Synquid, that will synthesize programs based on refinement types. At first glance this seems to be an entirely different approach than our tool, which instead uses examples as the specification and only uses refinement types as a search space pruning tool. However, the work in [11] does give an indication that our use of refinement types may be related on fundamental, proof theoretic level. By exploring this relationship in more depth, it may be possible to draw a stronger parallel between these various works and port ideas from one system to another.

One of the most widely used and well known instances of a programming by example system being used by many novice users is FlashFill[15]. Like other system, the goal of this work is to make executable, not code. This leaves users without the ability to modify generated source code. In the case of FlashFill, being embedding within production level software, most users are not clamoring for this feature. However it would certainly open an interesting avenue to introduce new users to computer programming if this were an option. StriSynth [16] takes a step in this direction by providing a natural language description of the synthesized program, but the source code remains obfuscated and inaccessible.

One difficult limitation is that without subexample generation we cannot recursively apply our algorithm as in the $Λ^2$ tool [9]. Subexample generation gives the ability to recursively call the synthesis engine to generate programs with multiple applications of high order functions. However, since the ability of $Λ^2$ to generate subexamples relied on hard coded subexample generation hypotheses for the predefined set of higher order functions, this does not scale. While inferring the hypotheses might be possible by inspecting the code, we have maintained a dedication to minimize our reliance on code analysis techniques for portability and longevity of the system. The best way to automatically create subexample generation functions solely based on type information remains a difficult problem.

9. Conclusions

Even for novice computer users, the need for basic programming skills is increasing. To meet this need, synthesis tools must be able to produce code that is not only correct, but also useful to the users of these tools. Program synthesis is powerful, but much of this power is left untapped when results are obfuscated by level upon level of folds, filters, and maps. Correctness is only a baseline, if program synthesizers do not produce natural code, most will not be able to take full advantage of this field.

There are many directions left to explore. Integrating an example-aware first-order function synthesis procedure is a clear next step, and enhancing our refinement type inference from examples will allow greater search space pruning for practical efficiency benefits. The offline / online structure of our algorithm allows novice users to interact with the useful synthesis part of our algorithm without dealing with the complicated refinement-type formulation of our search space pruning, so our tool would integrate well into a live programming environment.

Finally, as we saw section 7, our tool responds to synthesis queries with highly-readable, transparent, and ultimately
natural functions that fit the examples provided. We believe our tool is a step toward realizing a robust and accessible program synthesis system.
References


