SYNBIT: Synthesizing Bidirectional Programs using Unidirectional Sketches

MASAOMI YAMAGUCHI*, Graduate School of Information Sciences, Tohoku University, Japan
KAZUTAKA MATSUDA, Graduate School of Information Sciences, Tohoku University, Japan
CRISTINA DAVID, University of Bristol, UK
MENG WANG, University of Bristol, UK

We propose a technique for synthesizing bidirectional programs from the corresponding unidirectional code plus a few input/output examples. The core ideas are: (1) *constructing a sketch* using the given unidirectional program as a specification, and (2) *filling the sketch* in a modular fashion by exploiting the properties of bidirectional programs. These ideas are enabled by our choice of programming language, HOBiT, which is specifically designed to maintain the unidirectional program structure in bidirectional programming, and keep the parts that control bidirectional behavior modular. To evaluate our approach, we implemented it in a tool called SYNBIT and used it to generate bidirectional programs for intricate microbenchmarks, as well as for a few larger, more realistic problems. We also compared SYNBIT to a state-of-the-art unidirectional synthesis tool on the task of synthesizing backward computations.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Programming by example**; **Functional languages**.

Additional Key Words and Phrases: program synthesis, bidirectional transformation

ACM Reference Format:
Masaomi Yamaguchi, Kazutaka Matsuda, Cristina David, and Meng Wang. 2021. SYNBIT: Synthesizing Bidirectional Programs using Unidirectional Sketches. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 105 (October 2021), 31 pages. https://doi.org/10.1145/3485482

1 INTRODUCTION

Transforming data from one format to another is a common task of programming: compilers transform program text into syntax trees, manipulate the trees and then generate low-level code; database queries transform base relations into views; model-driving software engineering transforms one model into another. Very often, such transformations will benefit from being bidirectional, allowing changes to the targets to be mapped back to the sources too (for example the view-update problem in databases (Bancilhon and Spyratos 1981, Hegner 1990), bidirectional model transformation (Stevens 2008), and so on.).

As a response to this need, programming-language researchers started to design specialized programming languages for writing bidirectional transformations. In particular as pioneered by

*Currently at Fujitsu.

Authors' addresses: Masaomi Yamaguchi, masaomi.yamaguchi.t4@dc.tohoku.ac.jp, Graduate School of Information Sciences, Tohoku University, Sendai, Miyagi, Japan; Kazutaka Matsuda, kztk@ecei.tohoku.ac.jp, Graduate School of Information Sciences, Tohoku University, Sendai, Miyagi, Japan; Cristina David, cristina.david@bristol.ac.uk, University of Bristol, BS8 1QU, Bristol, Avon, UK; Meng Wang, meng.wang@bristol.ac.uk, University of Bristol, BS8 1QU, Bristol, Avon, UK.

2475-1421/2021/10-ART105
https://doi.org/10.1145/3485482

Pierce's group at Pennsylvania, a *bidirectional transformation* (BX), also known as a *lens* (Foster et al. 2007), is modeled as a pair of functions between source and view data objects, one in each direction. The forward function $get :: S \to V$ maps a source onto a view, and the corresponding backward function $put :: S \times V \to S$ reflects any changes in the view back to the source. Note that *get* is not necessarily injective. Accordingly *put*, in addition to the updated view, also takes the original source as an argument. This makes it possible to recover some of the source data that is not present in the view. Of course, not all pairing of *get*/*put* forms are valid BX; they must be related by specific properties known as *round-tripping*.

$$get\ s = v \quad \text{implies} \quad put\ (s, v) = s \qquad \textbf{(Acceptability)}$$
$$put\ (s, v) = s' \quad \text{implies} \quad get\ s' = v \qquad \textbf{(Consistency)}$$

for all $s, s' \in S$ and $v \in V$. Here, **Acceptability** states that no changes to the source happen if there is no change to the view, and **Consistency** states that all changes to the view must be captured in the updated source.

A BX language allows the transformations in both directions to be programmed together and is expected to guarantee round-tripping by construction.

This is a challenging problem for language design, and consequently compromises had to be made (in particular to usability) in favor of guaranteeing round-tripping. In the original lens design (Foster et al. 2007), lenses can only be composed by stylized lens combinators, which is inconvenient to program with. A lot of research has gone into this area since, for example Bohannon et al. (2008), Matsuda et al. (2007), Matsuda and Wang (2018b), Pacheco et al. (2014), Voigtländer (2009), and the state of the art has progressed a long way since. This includes a language HOBiT (Matsuda and Wang 2018b), which follows a line of research (Matsuda et al. 2007, Matsuda and Wang 2015a,b, Voigtländer 2009) that aims to produce BX code that is close in structure to how one will program the *get* function alone in a conventional unidirectional language. Despite the progresses in language design, BX programming is still considerably more difficult than conventional programming, especially when sophisticated backward behaviors are required. This complexity is largely inherent as one is asked to do more in less: defining behaviors in both directions in a single definition. Even in a language like HOBiT, where programmers are allowed (and indeed encouraged) to approach BX programming from the convenience of conventional unidirectional programming, there are still (necessary) additional code components that need to be added to the basic program structure to specify non-trivial backward behaviors.

Unidirectional program as sketch. In this paper we introduce Synbit, a program synthesis system that makes BX programming more approachable to mainstream programmers. In particular, we propose using unidirectional code (i.e., a definition of *get* in a Haskell-like language) as a sketch of the bidirectional program (which embodies both *get* and *put*). Consequently, programmers familiar with unidirectional programming can obtain bidirectional programs from unidirectional ones and input/output examples. In the neighboring field of software verification, expressing specifications (in our case sketches) as normal code has the effect of boosting the adoption of formal tools in industry (Chong et al. 2020), something that bidirectional programming research as a whole may benefit from.

It is not hard to see that this program sketch idea fits well with the language HOBiT. Unlike most BX languages, HOBiT is designed to keep bidirectional code as similar in structure as possible to how one may program the unidirectional *get*. Consequently, it is able to benefit from such a sketch and allow the synthesis process to mostly focus on parts of the code that specially handle intricate bidirectional behaviors. This is an attractive solution. On one hand, the specifications are intuitive: users simply write normal unidirectional programs (together with a few input/output

examples). On the other hand, the specifications as sketches are useful in the synthesis process because they reduce the search space. Moreover, this design supports gradual "bidirectionalization" done by incrementally converting existing unidirectional programs into bidirectional ones. As it will be shown in a comprehensive evaluation in Section 4, our system is highly effective and able to produce high-quality bidirectional programs in a wide range of scenarios.

Off-the-shelf synthesis is a non-solution. Before diving into the details of our proposed solution, we would like to take a step back and answer a question that may already be in some readers' minds: will program synthesis completely replace the need for bidirectional languages? That is, how about using generic synthesizers to derive a *put* from an existing *get* in a standard unidirectional language? After all, there already exist bidirectionalization techniques (Matsuda et al. 2007, Voigtländer 2009) that are able to derive a *put* from a *get* though in restricted situations.

When applied naively, this approach does not work. As an experiment, we tried using the state-of-the-art program synthesizer Smyth (Lubin et al. 2020) to generate the *put* from concrete examples and appropriate sketches. To simplify the problem, we ignored the round-tripping property between *get* and *put*, and tried to generate any *put* (even one that violates the laws). However, even in this simplified scenario, the synthesizer failed to find a *put* for simple examples (see Section 4.3 for more details).

This is not surprising because, while powerful, program synthesis is very hard due to the vast search space. The most common ways in which existing synthesis techniques circumvent this are by picking a reduced domain specific language to generate programs in (Gulwani 2011) and by seeding the program search with a sketch representing the program structure (Solar-Lezama 2009). In this paper, we are interested in synthesizing general purpose programs and therefore we do not adopt the first strategy.

Contributions:

- We present an application of program synthesis to the area of bidirectional programming. In particular, we provide an automated technique for generating bidirectional transformations in the language HOBiT (Section 3). The inputs to our procedure are the corresponding unidirectional code and a few concrete examples describing the backward transformation (Section 3.2).
- We exploit bidirectional programming properties, domain-specific knowledge of HOBiT and type information to efficiently prune the search space. In particular, we generate specialized program sketches from the unidirectional code (Section 3.3), which are then filled in a modular manner by separating the solving of dependent synthesis tasks (Sections 3.4 and 3.5).
- We present a classification of bidirectional programming benchmarks based on the amount of information from the source that is being lost through the forward transformation (Section 4.1). We believe that such a classification is valuable for evaluating the capabilities of our bidirectional synthesis technique.
- We implemented our bidirectional synthesis technique in a tool called Synbit, and used it to generate bidirectional programs for the set of benchmarks discussed above (Section 4). The prototype implementation of Synbit is available in the artifact [1] or the repository[2].

2 BACKGROUND: THE HOBiT LANGUAGE

HOBiT (Matsuda and Wang 2018b) is a state-of-the-art higher-order bidirectional programming language. A distinct feature of HOBiT is its support of a programming style that is close to the

[1] https://doi.org/10.5281/zenodo.5494504
[2] https://github.com/masaomi-yamaguchi/synbit

conventional unidirectional programming. The design of the language largely separates the core structure of programs (which can be shared with the unidirectional definition of *get*) from the specification of backward behaviors that are specific to bidirectional programming. In this section, we will introduce the core features of HOBiT with a focus on demonstrating its suitability as a target of sketch-based program synthesis. Curious readers who are interested in the full expressiveness power of HOBiT and the formal systems are encouraged to read the original paper (Matsuda and Wang 2018b).

2.1 A Simple Example

Before getting into HOBiT programs, we start with a familiar definition in Haskell below.

$$
\begin{array}{l}
\mathit{append} :: [a] \to [a] \to [a] \\
\mathit{append}\ \mathit{xs}\ \mathit{ys} = \textbf{case}\ \mathit{xs}\ \textbf{of}\ [\,] \quad \to \mathit{ys} \\
\qquad\qquad\qquad\qquad\qquad\quad a : x \to a : \mathit{append}\ x\ \mathit{ys}
\end{array}
$$

In the definition, we use explicit case branching (instead of syntax sugar in Haskell) to highlight the structure of the code.

Now, for a forward function (*get*) defined as *append*, let us investigate what will be suitable behaviors of its *put*. We denote the *put* by a HOBiT function $\mathit{appendB} :: \mathbf{B}[a] \to \mathbf{B}[a] \to \mathbf{B}[a]$. The **B**-annotated types (highlighted in blue) are *bidirectional types* in HOBiT, representing data that are subject to bidirectional computation. **B**-typed values are manipulated only by operations that satisfy the round-tripping laws, which is enough to ensure the round-tripping property of a whole program (Matsuda and Wang 2018b). As we will see in the sequel, bidirectional types can be mixed with normal unidirectional types to support flexible programming and greater expressiveness.

Bidirectional functions of type $\mathbf{B}\sigma \to \mathbf{B}\tau$ can be executed as bidirectional transformations between σ and τ in HOBiT's interactive environment (or, read-eval-print loop) via `:get` and `:put`. For example, one can run *appendB* forwards

```
> :get (uncurryB appendB) ([1, 2], [3, 4])
[1, 2, 3, 4]
```

and backwards.

```
> :put (uncurryB appendB) ([1, 2], [3, 4]) [5, 6, 7, 8]
([5, 6], [7, 8])
```

Note that we have *uncurried appendB* before execution by $\mathit{uncurryB} :: (\mathbf{B}a \to \mathbf{B}b \to \mathbf{B}c) \to \mathbf{B}(a, b) \to \mathbf{B}c$ so that it fits the pattern of $\mathbf{B}\sigma \to \mathbf{B}\tau$ for bidirectional execution. Specifically (*uncurryB appendB*) has type $\mathbf{B}([a], [a]) \to \mathbf{B}[a]$, and its *put* has type $([a], [a]) \to [a] \to ([a], [a])$.

Now we are ready to explore bidirectional behaviors.

2.1.1 Simple backward behavior. The simplest behavior of *put*, as adopted in Voigtländer (2009), is to only allow *in-place* update of views. In the case of *appendB*, it means that the changes to the length of the view list will result in an error.

```
> :put (uncurryB appendB) ([1, 2], [3, 4]) [5, 6, 7, 8]
([5, 6], [7, 8])
> :put (uncurryB appendB) ([1, 2], [3, 4]) [1, 2, 3]
Error: ...
```

To achieve this behavior, a definition in HOBiT reads the following.

```
appendB :: B[a] → B[a] → B[a]
appendB xs ys = case xs of [ ]    → ys
                           a : x → a : appendB x ys
```

As one can see, this definition is almost identical to that of *append* with only the language constructs such as case and data constructors being replaced by their bidirectional counterparts (underlined and highlighted in blue) that handle values of bidirectional types.

This simplicity comes from the design of HOBiT, as well as the modesty of the scenario. Given that the function is parametric in the list elements, in-place updates mean that the backward execution may simply trace back exactly the same control flow of the original forward execution. This can be achieved by recursing according to the original source (the first argument of *put*) and only using the updated view (the second argument of *put*) as a supplier of element values. Therefore, no additional specification is required in the code.

2.1.2 Branch switching. HOBiT is not limited to such simple behaviors. Its bidirectional language constructs seen above set us up for more sophisticated cases. Let's say that we now want to handle structural updates in the view, allowing the list length to vary.

```
> :get (uncurryB appendB) ([1, 2], [3, 4])
[1, 2, 3, 4]
> :put (uncurryB appendB) ([1, 2], [3, 4]) [5, 6, 7, 8]
([5, 6], [7, 8])
> :put (uncurryB appendB) ([1, 2], [3, 4]) [5, 6, 7, 8, 9]
([5, 6], [7, 8, 9])
> :put (uncurryB appendB) ([1, 2], [3, 4]) [5]
([5], [ ])
```

When the length of the view list changes, we try to change the second list of the source to accommodate that. If the length becomes shorter than that of the first source list, the second source list will be empty and the first source list will also change accordingly.

As one can see, this behavior can no longer be achieved by simply tracing back the original control flow of the forward execution. The backward execution will have to recurse a different number of times from the original, and how this is done will need to be additionally specified in the code. Here enters a definition in HOBiT that does exactly this.

```
appendB :: B[a] → B[a] → B[a]
appendB xs ys = case xs of [ ]    → ys                    with const True by λ_.λ_. [ ]
                           a : x → a : appendB x ys with not ∘ null  by λs.λ_. s
```

The code is longer than the last version, as expected, but the program structure remains the same: the additional specification for more sophisticated backward behavior is modularly grouped at the end of each case branch. Recall that we plan to use the unidirectional code as sketches to synthesize bidirectional code; this resemblance to the unidirectional code means that the synthesizing effort may now concentrate on the part specifying bidirectional behaviors, increasing its effectiveness.

In the above code, we used two distinctive HOBiT features known as *exit conditions* (marked by the with keyword) and *reconciliation functions* (marked by the by keyword). Both are for the purpose of controlling the backward behavior, especially when it no longer follows the original control flow (a behavior we call *branch switching*).

Exit conditions. An exit condition is an over-approximation of the forward-execution result of the branch, which always evaluates to True if the branch is taken (dynamically checked in HOBiT). Hence, an exit condition in a **case** expression has type $\tau \to \mathrm{Bool}$ if the whole **case** expression has type $\mathbf{B}\tau$. The exit conditions are then used as branching conditions in the backward execution. For example, in the above case of *appendB*, an empty list as view will choose the first branch, as the view does not match the condition $not \circ null$ of the second branch. Exit conditions often overlap; when multiple branches match, the original branch used in the forward execution is preferred. If impossible (as the exit condition of that branch does not hold), the topmost branch will be taken. Like the case of the in-place update we saw previously, if the view-list length is not changed, then in the backward execution of *appendB*, the exit conditions of the original branches (now used as branching conditions) are always satisfied, and therefore the original branches are always taken.

The situation becomes more interesting when the view update *does* change the length of the list, for example by making it shorter. In this case, the view list will be exhausted before the original number of recursions are completed. As a result, the backward execution will now see [ ] as its view input and a non-empty list as its source input. This means that the original branch at this point is the second branch, but the exit condition of that does not hold, which forces the first branch to be taken—a *branch switch*.

Reconciliation functions. We have seen that exit conditions may force branches to switch, which is crucial for handling interesting changes to the view. However, it only solves half of the problem; naive branch switching typically results in run-time failure. The reason is simple: when branch switching happens, the two arguments of *put* are in an inconsistent state for the branch; e.g., for *append*, having an non-empty source list (and an empty view list) is inconsistent for the branch $[] \to ys$. Reconciliation functions are used to fix this inconsistency. Basically, they are functions that take the inconsistent sources and views and produce new sources that are consistent with the branch taken. For example, in the definition above, the first branch will have [ ] as the new source, because a switch to this branch means an empty view and the branch expects the source to be the empty list for further *put* execution of the branch body. In general, a reconciliation function in a **case** expression is a function of type $\sigma \to \tau \to \sigma$, provided that the whole **case** expression has type $\mathbf{B}\tau$, with its scrutinee of type $\mathbf{B}\sigma$.

An interesting observation of this particular example of *append* is that the reconciliation function of the cons branch (i.e., $\lambda s.\lambda\_.s$ above) is actually never used. Recall that branch switching only happens when the backward execution tries to follow the original branch but the exit condition of the branch is not satisfied by the updates to the view. This will never happen in the nil branch above with the exit condition *const* True, which is always satisfied. In other words, regardless of the view update there will not be branch switching to the cons branch and therefore its reconciliation function is never executed. This behavior matches the behavior of *append* which recurses on the first source list: when the view list is updated to be shorter than the first source list, the recursion will need to be cut short (thus branch switching to the nil branch); but when the view list is updated to be longer, the additional elements will simply be added to the second source list, which does not affect the recursion (and thus no need of branch switching).

In summary, with reconciliation functions, the backward execution may recover from inconsistent states and resume with a new source. This is key to successful branch switching and the handling of structural updates to the view.

Round-tripping. It is also worth noting that branch switching in HOBiT does not threaten the round-tripping properties. Intuitively, the key principle of round-tripping is that a branch taken in a forward/backward execution should also be taken in a subsequent backward/forward execution (Foster et al. 2007, Hu and Ko 2016, Ko et al. 2016, Lutz 1986, Matsuda and Wang 2018b,

Yokoyama et al. 2011). When a brach switches in the backward execution, the new branch will produce a source value that matches the pattern of the new branch, ensuring that a subsequent forward execution will take the same branch. Since the exit conditions are checked as valid post conditions, this correspondence of forward/backward branchings is established, and consequently it guarantees round-tripping. An inappropriate reconciliation function will make the backward execution fail but not break round-tripping. More details can be found in the original paper (Matsuda and Wang 2018b). In this paper, we not only rely on the fact that HOBiT programs always satisfy round-tripping, but we also leverage the principle for effective synthesis (Section 3.5.2).

One can also observe that the exit conditions and reconciliation functions in *appendB* are quite simple themselves. However, their interaction with the rest of the code is intricate. Programmers who write them are therefore required to have a good understanding of how backward execution works and how it can be influenced, which may not come naturally. This combination of simplicity in form and complication in behavior makes it a fertile ground for program synthesis, which we set out to explore in this paper.

2.1.3 Mixing bidirectional and unidirectional programming. We end this section with another example of variants of *append*'s backward behavior and its implementation in HOBiT. The example also demonstrates a feature of HOBiT that supports a mixture of unidirectional and bidirectional programming for greater expressiveness. Let us consider the following definition.

```
appendBc :: B[a] → [a] → B[a]
appendBc xs ys = case xs of
    [ ]   → !ys                  with λv. length v == length ys by λ_.λ_. [ ]
    a : x → a : appendBc x ys with λv. length v ≠ length ys by λ_.λ(v : _). [v]
```

Noticeably, the type of the function is a mixture of bidirectional and unidirectional types, with the second argument as a normal list. Recall that bidirectional types represent data that are updatable; this type means that the second list is fixed with respect to backward execution. We will look at a few sample runs before going into the details of the definition. Note that since the second argument is constant in backward execution, there is no longer the need to uncurry the function; one can simply partially apply it as shown below.

```
> :get (λxs. appendBc xs ";") "apple"
"apple;"
> :put (λxs. appendBc xs ";") "apple" "pineapple;"
"pineapple"
> :put (λxs. appendBc xs ";") "apple" "plum;"
"plum"
```

In this case, the second list is `";"` and changes in the view can only affect the first list. Any attempt to change the last part of the view will (rightly) fail.

```
> :put (λxs. appendBc xs ";") "apple" "apple."
Error: ...
```

Now let us go back to the definition. The fact that the second argument *ys* is now of a normal (non-bidirectional) type means that it can be used in the exit conditions and reconciliation functions (which only involve unidirectional terms). During backward execution, the exit conditions dictate that the recursion will terminate (the first branch taken) when the view list is the same length as the original *ys*. In addition, since *ys* has a normal type, it will need to be *lifted* (as a constant) to the bidirectional world by ! so that the **case** expression becomes well typed. We again refer interested readers to Matsuda and Wang (2018b) for lifting in more general forms.

The mixture of unidirectional and bidirectional programming is a challenge to program synthesis as the search space has become much larger. Still, the fundamental has not changed: a definition of *get* remains a good sketch for HOBiT programs.

3 SYNTHESIS OF HOBIT PROGRAMS USING UNIDIRECTIONAL PROGRAMS AS SKETCHES

In this section, we describe our technique for synthesizing bidirectional programs in HOBiT. Throughout the section, we will use the familiar case of *append* as the running example.

3.1 Overview

Before presenting the technical details, we start with an informal overview of the synthesis process. Synbit takes in a unidirectional program (written in a subset of Haskell) and a small number of input/output examples of the required backward behavior, and produces a HOBiT program that behaves exactly as the input unidirectional program in the forward direction and is guaranteed to satisfy the round-tripping laws and conform to the given examples in the backward direction.

As an example, in the case of *append*, we provide the following specification to Synbit.

```
append :: [Int] → [Int] → [Int]
append = λxs. λys. case xs of {[ ] → ys; (a : x) → a : append x ys}
:put (uncurryB appendB) ([1, 2, 3], [4, 5]) [6, 2] = ([6, 2], [ ])
```

The definition of *append* above is completely standard. The user-provided input/output example specifies that the view list may be updated to a smaller length. As we have seen in Section 2, *append* needs to be uncurried before bidirectional execution, which is also reflected in the input/output example above where the source is a pair of lists. One interesting observation is that this bidirectional execution provides a call context of the function to be synthesized, which speeds up the synthesis process by narrowing down the choices of *appendB*'s type.

For the given specification, Synbit produces the following result.

```
appendB :: B[Int] → B[Int] → B[Int]
appendB = λxs. λys. case xs of {[ ] → ys
                                    with λv. case v of {x → True; _ → False}
                                    by   λs.λv. case v of {x → [ ]};
                               (a : x) → a : append x ys
                                    with λv. case v of {z : zs → True; _ → False}
                                    by   λs.λv. case v of {z : zs → s}}
```

As one can see, this program is equivalent to the hand-written definition in Section 2; the only difference is that the synthesized version does not use library functions such as *const* and *null*.[3]

Roughly speaking, the synthesis process that produces the above result involves two major components: the generation of a suitable sketch with holes and the filling of the holes. We will look at the main steps below.

Generation of sketches. The sketch is expected to be largely similar in structure to the unidirectional definition (thanks to the design of HOBiT), but there are a few details to be ironed out. First of all, one needs to decide the type of the target function. Recall that HOBiT is a powerful language that supports the mixing of unidirectional and bidirectional programming. Thus, for a type such as *append*'s, there are several possibilities such as B[Int] → B[Int] → B[Int], B[Int] → [Int] → B[Int], [BInt] → B[Int] → B[Int], and so on. It is therefore crucial to narrow

[3]Obvious cosmetic simplification could be made to part of the code for readability. But that is an orthogonal concern.

the choices down to control the search space. The call context in the input/output example(s) in the specification is useful for this step, as it can effectively restrict its type. We will discuss more details on this in Section 3.3. For now, it is sufficient to know that for the specification given in this example, the only viable type is $appendB :: \mathbf{B}[\mathsf{Int}] \to \mathbf{B}[\mathsf{Int}] \to \mathbf{B}[\mathsf{Int}]$.

The next step is to build a sketch based on the unidirectional definition given in the specification. The type we have from above straightforwardly implies that the **case** construct in *append*'s definition is to be replaced by the bidirectional **case**, which expects exit conditions and reconciliation functions to be added (as holes (□) in the sketch).

$$\begin{aligned} appendB = \lambda xs.\, \lambda ys.\, \underline{\mathbf{case}}\ xs\ \underline{\mathbf{of}}\ \{[\,] &\to ys\ \underline{\mathbf{with}}\ \square\ \underline{\mathbf{by}}\ \square; \\ (a : x) &\to a : appendB\ x\ ys\ \underline{\mathbf{with}}\ \square\ \underline{\mathbf{by}}\ \square\} \end{aligned}$$

Both the exit conditions and reconciliation functions are simply unidirectional functions. Thus in theory, one can try to use a generic synthesizer to generate them. However, this naive method will miss out on a lot of information that we know about these functions. Recall that, given a **case** branch $p \to e$, its corresponding exit condition must return true for all possible evaluations of e; similarly, the results of its reconciliation function must match p and the second argument of the reconciliation function must be an evaluation result of e. We therefore capture such knowledge with specialized sketches, which make use of two types of specialized holes that are parameterized with additional information: *exit-condition hole* ($\square^{\mathrm{e}}(e)$), and *reconciliation-function hole* ($\square^{\mathrm{r}}(p, e)$). This results in the following sketch for this example.

$$\begin{aligned} appendB = \lambda xs.\, \lambda ys.\, \underline{\mathbf{case}}\ xs\ \underline{\mathbf{of}}\ \{[\,] &\to ys\ \underline{\mathbf{with}}\ \square^{\mathrm{e}}(ys)\ \underline{\mathbf{by}}\ \square^{\mathrm{r}}([\,], ys); \\ (a : x) &\to a : append\ x\ ys \\ &\quad\ \underline{\mathbf{with}}\ \square^{\mathrm{e}}(a : append\ x\ ys) \\ &\quad\ \underline{\mathbf{by}}\ \ \square^{\mathrm{r}}((a : x), a : append\ x\ ys)\} \end{aligned}$$

In the spirit of component-based synthesis (Feng et al. 2017, Jha et al. 2010), we generate code by composing components from a library that includes **case** and **case** expressions, Bool constructors and operators, as well as list and tuple constructors. As we will explain in Section 3.2, this library can be augmented with auxiliary components provided by the user.

In this example, the sketch generation is quite deterministic. In general, especially when multiple functions must be synthesized together and auxiliary components are provided, there could be multiple candidate sketches. In such a case, we use a lazy approach that nondeterministically tries exploring one candidate and generating any other.

Sketch completion step I: shape-restricted holes. With the sketch ready, we can proceed to fill the holes. As a first step in the sketch completion process, we make use of the information captured by the specialized holes to generate some parts of the code for exit conditions and reconciliation functions. This step does not involve any search.

$$\begin{aligned} appendB = \lambda xs.\, \lambda ys.\, \underline{\mathbf{case}}\ xs\ \underline{\mathbf{of}}\ \{[\,] &\to ys\ \underline{\mathbf{with}}\ \lambda v.\, \mathbf{case}\ v\ \mathbf{of}\ \{x \to \square;\ \_ \to \mathsf{False}\} \\ &\quad\ \underline{\mathbf{by}}\ \ \lambda s.\lambda v.\, \mathbf{case}\ v\ \mathbf{of}\ \{x \to \square([\,])\}; \\ (a : x) &\to a : append\ x\ ys \\ &\quad\ \underline{\mathbf{with}}\ \lambda v.\, \mathbf{case}\ v\ \mathbf{of}\ \{z : zs \to \square;\ \_ \to \mathsf{False}\} \\ &\quad\ \underline{\mathbf{by}}\ \ \lambda s.\lambda v.\, \mathbf{case}\ v\ \mathbf{of}\ \{z : zs \to \square(a : x)\}\} \end{aligned}$$

The specialized holes are replaced with λ-abstractions with **case** structures. The result involves a different type of holes we call *shape-restricted holes* ($\square(p)$); such holes can only be filled with expressions that may match the pattern p. For example, for $\square(a : x)$, the empty list is not a valid

candidate. A generic hole (□) is a special case where the pattern is a wildcard that matches every term.

For exit conditions, the translation used the information encoded by exit condition holes to figure out when False should be returned—recall that, for a **case** branch $p \to e$, exit conditions should return False for any results that cannot be produced by e. In the case of *appendB*, this means that for the second branch in the sketch, the exit condition must return False for any empty list. (Here, z and zs are fresh variables.) For the first branch, this information does not help us eliminate any candidates. (Again, x is a fresh variable.) The **case** construct generation uses all the information encoded by the exit condition holes. Consequently, the holes left in the sketch are generic ones.

For reconciliation functions, the newly generated shape-restricted holes capture the fact that for a **case** branch $p \to e$, the result of the reconciliation function must match p. Thus, the first branch of *appendB* has $\square([\,])$ while the second one $\square(a : x)$. We further know that the second argument of the reconciliation function must be a result of e, which allows us to generate the **case** structure shown in the sketch.

Sketch completion step II: search and filtering. The last step is to fill the remaining shape-restricted holes. At this point, we leave off using the information in the unidirectional input program, and turn our attention to the input/output example(s). To fill the holes, we generate β-normal forms where functions are η-expanded, and filter the candidates by checking against the examples(s). A problem with using the example(s) to filter out incorrect candidates is that it is for the whole program, which includes several holes. A naive use of the example(s) means that filtering has to be delayed until late in the synthesis process when all the holes are filled. This is inefficient.

Conversely, our ideal goal is to have a modular filtering process, where we can simultaneously check candidate exit conditions and reconciliation functions independently of each other. For this purpose, our solution is to leverage domain-specific knowledge of HOBiT. Specifically, we make use of the fact that *put* (s, v) and *get* (*put* (s, v)) must follow the same execution trace in terms of taken branches, as explained in the discussion on round-tripping in HOBiT (see the corresponding paragraph in Section 2). This enables us to fix the control flow of the *put* behavior for the given input/output example(s) without referring to exit conditions, so that we can separate the search for exit conditions from reconciliation functions. We will discuss this in more detail later in the overview, as well as in Section 3.5.

Moreover, the use of the trace information also enables us to address the issue of non-terminating *put* executions. In a naive generate-and-test synthesis approach, some of the generated candidates may be non-terminating, which poses issues for the testing phase. As we assume that the *put* execution must always follow the finite branching trace of *get*, we never generate such programs. Here, we assumed that the input/output unidirectional program is terminating for the original and updated sources of the input/output examples. More details on this will be presented in Section 3.5.2.

Filtering of exit conditions based on branch traces. We continue with the partially filled sketch for *appendB* above (reproduced below), with the holes numbered for easy reference.

$$\begin{aligned} \mathit{appendB} = \lambda xs.\, \lambda ys.\, \underline{\textbf{case}}\ xs\ \underline{\textbf{of}}\ \{[\,] \to ys\ \underline{\textbf{with}}\ \lambda v.\, \textbf{case}\ v\ \textbf{of}\ \{x \to \square_1;\ \_ \to \mathsf{False}\} \\ \underline{\textbf{by}}\quad \lambda s.\lambda v.\, \textbf{case}\ v\ \textbf{of}\ \{x \to \square([\,])_3\}; \\ (a : x) \to a : \mathit{append}\ x\ ys \\ \underline{\textbf{with}}\ \lambda v.\, \textbf{case}\ v\ \textbf{of}\ \{z : zs \to \square_2;\ \_ \to \mathsf{False}\} \\ \underline{\textbf{by}}\quad \lambda s.\lambda v.\, \textbf{case}\ v\ \textbf{of}\ \{z : zs \to \square(a : x)_4\}\} \end{aligned}$$

What are the constraints on the holes that we can derive from the input/output example below?

$$\texttt{:put}\ (\mathit{uncurryB}\ \mathit{appendB})\ ([1, 2, 3], [4, 5])\ [6, 2] = ([6, 2], [])$$

As mentioned above, `:put` (*uncurryB appendB*) ([1, 2, 3], [4, 5]) [6, 2] must choose the branches chosen by `:get` (*uncurryB appendB*) ([6, 2], []). We shall call a history of chosen branches a *branch trace*. For `:get` (*uncurryB appendB*) ([6, 2], []), the branch trace is:

(i) the cons branch (where *xs* is [6, 2]),
(ii) the cons branch (where *xs* is [2]),
(iii) the nil branch (where *xs* is []).

We now follow the same trace for `:put` ((*uncurryB appendB*) ([1, 2, 3], [4, 5]) [6, 2]) and each step will give rise to a constraint on the exit condition of the branch.

(i) a : *append x ys* (and therefore the v) has the value of the update view [6, 2], and $\square_2$ must evaluate to True in this context. Therefore, $\square_2[6/z, [2]/zs, [6, 2]/v] \equiv$ True.
(ii) a : *append x ys* (and therefore the v) has the value of the update view [2], and $\square_2$ must evaluate to True in this context. Therefore, $\square_2[2/z, [\,]/zs, [2]/v] \equiv$ True.
(iii) *ys* (and therefore the v) has the value of [ ], and $\square_1$ must evaluate to True in this context. Therefore, $\square_1[[\,]/x, [\,]/v] \equiv$ True.

These constraints are useful in generating the exit conditions independently. As a matter of fact, in the case of *appendB* both $\square_1$ and $\square_2$ are simply filled by the expression True which satisfies all the constraints.

There are no trace constraints generated for holes 3 and 4 though. So they will be generated according to the shape restrictions only. Hole 3 must be [ ] while Hole 4 can be filled by a non-empty list. Recall that in this example, the reconciliation functions of the cons branch are never used. And therefore, arbitrary default terms will fill Hole 4 just fine, which produces the output we saw at the beginning of this subsection.

Filtering of reconciliation functions based on branch traces. The branch traces are also used to filter reconciliation functions. (This is not needed in this example as the nil branch was already fixed in the filling of shape-restricted holes and the cons branch can be arbitrary.) The important insight here is that reconciliation functions can be filtered independently from the exit conditions, resulting in significant efficiency gain. The reason is that the branch traces carry all the information that is needed to test reconciliation functions (recall that the exit conditions are only for determining branching in backward execution; and since the branching is known in the branch traces there is no need for exit conditions.). We will see examples of this in Section 3.5.2.

In the rest of this section, we will go through each step of the synthesis process in detail.

3.2 Input to Our Method

Remember from the overview that our technique takes as input some typed unidirectional code and a set of input/output examples illustrating the backward transformation. Formally, this translates to the following 4-tuple $I = (P, \Gamma, f_1, \mathcal{E})$:

- $P = \{f_i = e_i\}_i$ is a program in the unidirectional fragment of HOBiT (a subset of (strict) Haskell), where $f_i = e_i$ stands for a function/value definition of f_i by the value of e_i.
- $\Gamma = \{f_i : A_i\}_i$ is a typing environment for P; i.e., each e_i has type A_i under Γ.
- f_1 is the entry point function, whose type is expected to have the form $\sigma_1 \to \tau_1$;[4] this is used to prune the search space as explained in Section 3.3.
- $\mathcal{E} = \{(s_k, v_k, s'_k)\}_k$ is a (finite) set of well-typed input/output examples for a bidirectional version of the entry point f_1.

[4]We use metavariables $A, B, \ldots$ for types in general and $\sigma, \tau, \ldots$ for those that can be sources or views. In Matsuda and Wang (2018b), the latter kind of types do not contain **B** and $\to$, but their implementation does not distinguish the two (which in fact is safe). Thus, we do not strictly respect the restriction on σ-types in our technical development.

The input program P may contain functions that are not reachable from the entry point but can be used during program generation. We call such functions *auxiliary functions* and add them to our library of default synthesis components. As mentioned earlier in Section 3.1, the default library includes **case** and **case** expressions, Bool constructors and operators, as well as list and tuple constructors.

Example 3.1 (append). For the *appendB* example, the input is formally expressed as:

$$P_{\text{app}} = \{appendB = \lambda xs.\, \lambda ys.\, \textbf{case}\ xs\ \textbf{of}\ \{[\,] \to ys;\ (a : x) \to a : appendB\ x\ ys, uncurryB = \ldots\}\}$$
$$\Gamma_{\text{app}} = \{appendB : [\mathsf{Int}] \to [\mathsf{Int}] \to [\mathsf{Int}], uncurryB : \ldots\}$$
$$f_{1\text{app}} = uncurryB\ appendB$$
$$\mathcal{E}_{\text{app}} = \{(([1,2,3],[4,5]),[6,2],([6,2],[]))\}.$$

Here, we omit the definition and the type of *uncurryB* but state it is a part of the input program. □

3.3 Generation of Sketches

As shown in the overview, we start by generating bidirectional sketches from the unidirectional code. The basic idea of the sketch generation is to replace unidirectional constructs with bidirectional ones nondeterministically: when **case** is replaced with **case**, exit conditions and reconciliation functions are left as holes. Interestingly, replacing all unidirectional constructs (if they have corresponding bidirectional ones) may not be the best solution; as demonstrated in $appendBc :: \mathbf{B}[a] \to [a] \to \mathbf{B}[a]$ in Section 2, we sometimes need to leave some parts unidirectional to achieve given bidirectional behavior.

The starting point of sketch generation is deciding the type of the target function. We expect the unidirectional code to contain an entry point function $f_1 : \sigma_1 \to \tau_1$ (e.g., *uncurry append* in the Example 3.1). This helps us reduce the number of generated type signatures as we know that the target entry point function to be synthesized must have type $\mathbf{B}\sigma_1 \to \mathbf{B}\tau_1$. Also, we further prune the search space by eliminating type signatures that do not obey the call context in the input/output examples.

Type signature generation. We first define the relation $A \rightsquigarrow A'$ as: A' is the type obtained from A by replacing an arbitrary number of sub-components σ in A by $\mathbf{B}\sigma$ nondeterministically, as long as σ does not contain function types. We do not replace σ containing function types to avoid generating apparently non-useful types such as $\mathbf{B}(\mathsf{Int} \to \mathsf{Int})$ and $\mathbf{B}[\mathsf{Int} \to \mathsf{Int}]$. Next, we provide the typing environment generation relation $\Gamma \rightsquigarrow \Gamma'$, where Γ' is the typing environment corresponding to the bidirectional program.

Definition 3.2 (Generation of Typing Environment). For $\Gamma = \{f_1 : \sigma_1 \to \tau_1\} \cup \{f_i : A_i\}_{i>0}$, the typing environment generation relation $\Gamma \rightsquigarrow \Gamma'$ is defined if $\Gamma' = \{f_1 : \mathbf{B}\sigma_1 \to \mathbf{B}\tau_1\} \cup \{f_i : A'_i\}_{i>0}$, where $A_i \rightsquigarrow A'_i$ for each $i > 0$. □

Type-directed sketch generation. Once we have (a candidate) typing environment Γ', the next step is to generate corresponding sketches in a type-directed manner. Very briefly, the type system in HOBiT (Matsuda and Wang 2018b) uses a dual context system (Davies and Pfenning 2001). The typing relation can be written as $\Gamma; \Delta \vdash e : A$, where Γ and Δ respectively are called unidirectional and bidirectional typing environments, and hold variables introduced by unidirectional and bidirectional contexts respectively.

The sketch generation is done by using a relation $\Gamma'; \Delta'; A' \triangleright \Gamma \vdash e : A \rightsquigarrow e'$, which reads that, from a term-in-context $\Gamma; \emptyset \vdash e : A$, sketch e' is generated according to the given target typing environments Γ' and Δ', and target type A' so that $\Gamma'; \Delta' \vdash e' : A'$ holds after the sketch has been

completed (i.e., no unfilled holes) in a type-preserving way. Notice that Γ', Δ' and A' are also a part of the input in $\Gamma'; \Delta'; A' \triangleright \Gamma \vdash e : A \rightsquigarrow e'$; i.e., its outcome is only e'. We omit the concrete generation rules due to the space limitation, and put them in the extended version of this paper (Yamaguchi et al. 2021).

We note that the rules are overlapping (i.e., several may be applicable at a given step), which makes sketch generation nondeterministic. The sketch generation is defined formally as below.

Definition 3.3 (Type-Directed Sketch Generation). Suppose that $\Gamma \rightsquigarrow \Gamma'$. Then, for $P = \{f_i : e_i\}_i$, the sketch generation relation $P \rightsquigarrow P'$ is defined if $P' = \{f_i : e'_i\}_i$, where $\Gamma'; \emptyset; \Gamma'(f_i) \triangleright \Gamma \vdash e_i : \Gamma(f_i) \rightsquigarrow e'_i$. □

3.4 Sketch Completion Step I: Shape-Restricted Holes

In general, there will be several possible sketches for a given unidirectional program. As mentioned in Section 3.1, in such a case, we use a lazy approach that nondeterministically tries exploring one candidate and generating any other. In this section we describe the sketch exploration process. In particular, we start by using the information captured by the specialized holes to generate parts of the code for exit conditions and reconciliation functions.

3.4.1 Handling exit condition holes. Remember that an exit condition matching a hole $\square^{\mathrm{e}}(e)$ should return False for any results that cannot be produced by e. Then, our idea here is to generate code that returns False for values that are obviously not the result of e. For example, for $\square^{\mathrm{e}}(a : \mathit{append}\ x\ ys)$, we generate code returning False for the empty list.

Let us write $\mathcal{P}(e)$ for a pattern that represents an obvious shape of e, defined as follows (where C is a constructor):

$$\mathcal{P}(e) = \begin{cases} \mathsf{C}\ \mathcal{P}(e_1)\ \ldots\ \mathcal{P}(e_n) & \text{if } e = \mathsf{C}\ e_1\ \ldots\ e_n \\ x & \text{otherwise } (x : \text{fresh}) \end{cases}$$

For example, we have $\mathcal{P}(a : \mathit{append}\ x\ ys) = \mathcal{P}(a) : \mathcal{P}(\mathit{append}\ x\ ys) = z : zs$, where z and zs are fresh, conforming to the second case above. It is quite apparent that any result of e matches with $\mathcal{P}(e)$; in other words, values that do not match with $\mathcal{P}(e)$ cannot be a result of e. Using $\mathcal{P}(e)$, we concretize exit-condition holes as below.

Definition 3.4 (Partial completion of exit condition holes). Let p_e be a pattern $\mathcal{P}(e)$. Then, the exit-condition-hole partial completion relation $\square^{\mathrm{e}}(e) \rightsquigarrow e'$, which reads hole $\square^{\mathrm{e}}(e)$ is filled by e', is defined by the rule

$$\square^{\mathrm{e}}(e) \rightsquigarrow \lambda s.\ \mathbf{case}\ s\ \mathbf{of}\ \{p_e \to \square;\ \_ \to \mathsf{False}\}$$ □

Note that the resulting sketch will contain a generic hole □, whose shape is no longer constrained. For example, $\square^{\mathrm{e}}(a : \mathit{append}\ x\ ys)$ is converted as follows

$$\square^{\mathrm{e}}(a : \mathit{append}\ x\ ys) \rightsquigarrow \lambda s.\ \mathbf{case}\ s\ \mathbf{of}\ \{z : zs \to \square;\ \_ \to \mathsf{False}\}$$

3.4.2 Handling reconciliation function holes. Remember that the role of a reconciliation function associated with a branch is to reconcile the original source with the branch by producing a new "original source" matching the branch (Section 2). Thus, when the branch has the form $p \to e$, the reconciliation function must return a value of the form $p[\overline{v/x}]$ where $\{\overline{x}\} = \mathrm{fv}(p)$. Hence, a natural approach is to generate reconciliation functions of the form $\lambda s.\lambda\mathcal{P}(e).p[\overline{e/x}]$.

However, only considering expressions of the aforementioned form limits the use of user-specified auxiliary functions in reconciliation functions. Instead, we generate reconciliation functions of the form $\lambda s.\lambda\mathcal{P}(e).\square(p)$. Recall that the shape-restricted hole $\square(p)$ will be filled by expressions shaped p. This idea is formally written as below.

Definition 3.5 (Partial completion of reconciliation function holes). Let p_e be a pattern $\mathcal{P}(e)$. Then, the partial completion relation for the reconciliation function hole, $\square^{\mathrm{r}}(p, e) \rightsquigarrow e'$, which reads hole $\square^{\mathrm{r}}(p, e)$ is filled by e', is defined by the rule

$$\square^{\mathrm{r}}(p, e) \rightsquigarrow \lambda s.\lambda v.\, \mathbf{case}\ v\ \mathbf{of}\ \{p_e \to \square(p)\} \qquad \square$$

3.5 Sketch Completion Step II: Search and Filtering

The last step is to fill the remaining shape-restricted and generic holes. This process involves type-directed generation of candidates and filtering based on user-provided input/output examples. For simplicity of presentation, we do not explicitly capture the type of the code to be generated in the shape-restricted holes; instead, we recover it from the sketch and typing environment Γ.

3.5.1 Generating candidates for shape restricted holes. In this section, we describe the process of filling in shape restricted holes $\square(p)$. To achieve this, we generate terms of shape p in β-normal forms where functions are η-expanded. Specifically, we produce expressions U^p in the following grammar, which restricts their shape to p:

$$\begin{array}{lll} U^p ::= V^p \mid \mathbf{case}\ x\ V_1\ \ldots\ V_n\ \mathbf{of}\ \{p_i \to U_i^p\}_i & & \\ V^p ::= \lambda x.U & & (p = x) \\ \quad \mid\ x\ V_1\ \ldots\ V_n & & \\ \quad \mid\ \mathsf{C}\ V_1^{p_1}\ \ldots\ V_n^{p_n} & & (p = \mathsf{C}\ p_1\ \ldots\ p_n \text{ or } p, p_1, \ldots, p_n \text{ are all variables}) \end{array}$$

To simplify the presentation, we omit p and write V or U if p is a variable. In this grammar, the purpose of U is to have **case**s in the outermost positions (but inside λ); a **case** in a context $K[\mathbf{case}\ e\ \mathbf{of}\ \{p_i \to e_i\}_i]$ can be hoisted as $\mathbf{case}\ e\ \mathbf{of}\ \{p_i \to K[e_i]\}_i$, which is a transformation known as commuting conversion. If p is not a variable, we can only generate constructors as specified by p ($p = \mathsf{C}\ p_1\ \ldots\ p_n$ or $(p, p_1, \ldots, p_n)$ are all variables). Otherwise, if p is a variable, the only knowledge we assume about it is its type. Thus, any of the productions for V^p would be considered. Note that x/C are drawn from the current context; i.e., they may be components provided by users.

Types are used for two purposes in this type-directed generation. The rather obvious purpose is to limit the search space for x and C; notice that, since we know their types, we also know the types of their arguments allowing us to perform type-directed synthesis for them as well. The other purpose is to reduce redundancy with respect to η-equivalence by only generating $\lambda x.U$ for function types and **case**s only for non-function types. A caveat is the generation of $x\ V_1\ \ldots\ V_n$ at the scrutinee position of **case**, which cannot be done in a type-directed way as its type is not given *a priori*; instead, its type is synthesized by using the type of x.

3.5.2 Filtering based on branch traces. As explained in the discussion on round-tripping in HOBiT (see the corresponding paragraph in Section 2), we leverage the fact that $put\ (s, v)$ and $get\ (put\ (s, v))$ must follow the same execution trace in terms of taken branches. This enables us to fix the control flow of the *put* behavior for the given input/output example(s) without referring to exit conditions, making it possible to separate dependent synthesis tasks.

While the example *appendB* in Section 3.1 only showed how filtering works for exit conditions, we provide next one example illustrating filtering based on branch traces for both exit conditions and reconciliation functions. We conclude with a discussion on pruning away programs that would otherwise cause non-terminating *put* executions. See **??** for the formal descriptions of this trace-based filtering process.

Example of filtering exit conditions and reconciliation functions based on branch traces. Consider the following program

$$f :: \text{Either Int Int} \to \text{Bool}$$
$$f\ x = \textbf{case}\ x\ \textbf{of}\ \{\text{Left}\ x \to \text{True};\ \text{Right}\ x \to \text{False}\}$$

that comes with two input/output examples that negate the view and cause the sources to flip: $\mathcal{E} = \{(\text{Left } 42, \text{False}, \text{Right } 42),\ (\text{Right } 42, \text{True}, \text{Left } 42)\}$. Suppose that from the given unidirectional code, we obtain the following candidate before any filtering is done.

$$\begin{aligned} f\ x = \underline{\textbf{case}}\ x\ \underline{\textbf{of}}\ \{\text{Left}\ x \to \underline{\text{True}} \quad & \underline{\textbf{with}}\ \lambda v.\textbf{case}\ v\ \textbf{of}\ \{\text{True} \to \square_1;\ \text{False} \to \text{False}\} \\ & \underline{\textbf{by}}\quad \lambda s.\lambda v.\textbf{case}\ v\ \textbf{of}\ \{\text{True} \to \square_2\}; \\ \text{Right}\ x \to \underline{\text{False}} \quad & \underline{\textbf{with}}\ \lambda v.\textbf{case}\ v\ \textbf{of}\ \{\text{False} \to \square_3;\ \text{True} \to \text{False}\} \\ & \underline{\textbf{by}}\quad \lambda s.\lambda v.\textbf{case}\ v\ \textbf{of}\ \{\text{False} \to \square_4\}\} \end{aligned}$$

We first discuss filtering of exit conditions. For the first example, `:get` f (Right 42) takes the second branch, meaning that we obtain the constraint $\square_3[\text{False}/v] \equiv \text{True}$. For the second example, `:get` f (Left 42) takes the first branch, generating the constraint $\square_1[\text{True}/v] \equiv \text{True}$. A solution for these constraints is $\square_1 = \text{True}$ and $\square_3 = \text{True}$.

Now, let us focus on the reconciliation functions. If we consider the branch trace generated by the *get* and evaluate the *put* for the given examples, we obtain the following constraints:

- For `:put` f (Left 42) False, we must switch branches to the second branch, meaning that the reconciliation function corresponding to the second branch gets triggered, generating the constraint: `:put` $f(\square_4[\text{False}/v])$ False = Right 42.
- For `:put` f (Right 42) True, we must switch branches to the first branch, meaning that the reconciliation function corresponding to this branch gets triggered, generating the constraint: `:put` $f(\square_2[\text{True}/v])$ True = Left 42.

From these constraints, one possible solution is $\square_2 = \text{Left } 42$ and $\square_3 = \text{Right } 42$. While this solution obeys the given example, a better one would be $\square_2 = \textbf{case}\ s\ \textbf{of}\ \{\text{Left}\ x \to s; \text{Right}\ y \to \text{Left}\ y\}$ and $\square_3 = \textbf{case}\ s\ \textbf{of}\ \{\text{Left}\ x \to \text{Right}\ x; \text{Right}\ y \to s\}$; each s in the branch bodies in $\square_2$ and $\square_3$ can be arbitrary, as they will never be used. The suboptimal solution could be filtered out by our synthesis engine if other examples such as `:put` f (Left 37, False) = Right 37 were provided by the user.

As a note, for both the previous example and the running example *appendB* in Section 3.1, we only generate positive constraints (i.e., that evaluate to True) for the holes in exit conditions. However, in certain cases, negative constraints (i.e., that evaluate to False) may also be generated. This happens when branch switching implies that the original branch's exit condition evaluates to False. We encountered such situations for *lengthTail* and *reverse* in Section 4. In such a case, the choice of reconciliation functions may affect the generated constraints as they specify new sources.

Discussion on pruning non-terminating programs based on branch traces. Using branch traces also helps us prune away programs that would cause non-terminating *put* executions. It is known that synthesis of recursive functions is a challenging problem (Albarghouthi et al. 2013), especially for programming-by-examples, because a synthesized function may diverge for a given example. Waiting for a timeout is inefficient and there is no clear way to set an appropriate time limit. In our approach, assuming that the given *get* execution is terminating for the input/output examples, we never generate such diverging candidate programs. The reason is that we only generate programs whose *put* execution follows the finite branch trace of the *get*, and are thus terminating.

3.6 Heuristics

In this section, we discuss some heuristics we found effective when exploring the search space.

Assigning costs to choices. Our generation is prioritized by assigning a positive cost to each nondeterministic choice in the sketch generation. Programs with lower costs are generated earlier than those with higher costs. An advantage of this approach is that it is easy to integrate with lazy nondeterministic generation methods (Fischer et al. 2011), which is the core of our prototype implementation. Another advantage is that smaller programs naturally have higher priority (i.e., lower costs), as the generation of large programs usually involves many choices, reflecting our belief that smaller programs are typically preferable.

Canonical forms of Bool*-typed expressions.* Generation of Bool-typed expressions is a common task, especially in our context as exit conditions always return Bool values. However, a naive generation of Bool-typed expressions may lead to redundancies, for example, True && e and e may be considered two distinct expressions during the search. So when filling holes of type Bool, we generate expressions in disjunctive normal form, in which atomic propositions are expressions of the form $x\ V_1\ \ldots\ V_n$ with $x : A_1 \to \cdots \to A_n \to \mathsf{Bool} \in \Gamma$. While this eliminates redundancy due to distributivity, associativity and zero and unit elements, it does not address commutativity and idempotence. Provided that there is a strict total order $\prec$ on expressions, both sorts of redundancy could be addressed easily by generating e_2 *after* e_1 so that $e_1 \prec e_2$ holds. Currently we do not do this in our implementation in order to avoid the additional overhead of checking $e_1 \prec e_2$.

Other effective improvements. In addition to the heuristics mentioned above, we make use of some simple but effective techniques. For example, for **case** with a single branch, we do not try to synthesize exit conditions or reconciliation functions. An exit condition $\lambda\_.\mathsf{True}$ and a reconciliation function $\lambda s.\lambda v.s$ suffice for such a branch. We do not generate redundant case expressions such as $\lambda s.\lambda v.\mathbf{case}\ v\ \mathbf{of}\{z \to \cdots\}$. When the pattern p of a branch does not contain any variables, we deterministically choose $\lambda\_.\lambda\_.p$ as its reconciliation function. For a **case** whose patterns $\{\mathcal{P}(e_i)\}_i$ do not overlap, we do not leave holes in exit conditions as replacing them with True is sufficient.

3.7 Soundness and Incompleteness

Our proposed method is sound for the given input/output examples in the sense that it synthesizes a bidirectional transformation such that its *put* behavior is consistent with the input/output examples, and its *get* behavior coincides with the given *get* program for the sources that appear in the examples. This is obvious because we check the conditions in the last step (i.e., filtering) in our synthesis. It is worth noting that the *get* behavior of a synthesized function may be less defined than a given *get* program, because our method may synthesize exit conditions that are not postconditions; recall that they are checked dynamically in HOBiT (Section 2). We heuristically try to avoid this by prioritizing True over False in the synthesis of exit conditions, which works effectively for all the cases discussed in Section 4 but is not a guarantee, especially with components. We could address this by inferring postconditions and using them as exit conditions, which is left for future work.

In contrast, our proposed method is incomplete. This is due to the use of the sketches obtained from the unidirectional code to prune the search space. While this makes our approach efficient, it may remove potential solutions. Such situations are captured by the examples *lines* and *lookup*, where the solutions do not follow the sketches, in the experimental evaluation in Section 4.1.

4 EXPERIMENTS

We implemented the proposed idea as a proof-of-concept system, Synbit, in Haskell[5]. Synbit is given as an extension to the original HOBiT implementation (Matsuda and Wang 2018b).—-z———————

[5]The implementation is available in the artifact https://doi.org/10.5281/zenodo.5494504 or in https://github.com/masaomi-yamaguchi/synbit

We measure the effectiveness of our proposed method in the following three experiments.

- Microbenchmarks, classified in terms of information loss (Section 4.1).
- More realistic problems including XML transformations and string parsing (Section 4.2).
- Comparisons with the other state-of-the-art synthesis methods (Section 4.3).

The experiments were conducted on a Windows Subsystem for Linux (WSL) 2 running on a laptop PC with 2.30 GHz Intel(R) Core(TM) i7-4712HQ CPU and 16 GB memory, 13 GB out of which were assigned to WSL 2. The host OS was Windows 10 (build NO. 19042.685), and the guest OS was Ubuntu 20.04.1 LTS. We used GHC 8.6.5 to compile Synbit with the optimization flag `-O2`. Execution times were measured by Criterion[6], a popular library in Haskell for benchmarking, which estimates the true execution time by the least-squares method. Any case running longer than 10 minutes was reported as a timeout.

4.1 Microbenchmarks Classified by Information-Loss

To construct the microbenchmarks, we classify programming problems according to the level of difficulty. Recall that the main challenge of BX is to incorporate the information that is in the source but absent in the view in order to create an updated source. For structure rich data represented by algebraic datatypes, this includes the structure of the source data, especially the part that the *get* function recurs on. With that, we arrive at the following classes.

Class 1 All information of the recursion structure is present in the view (e.g., *map*).
Class 2 Some information of the recursion structure is present in the view (e.g., *append*).
Class 3 No information of the recursion structure is present in the view (e.g., *lookup*).

The rule of thumb is that the more information is present in the view, the easier is it to define a *put* that handles structural changes to the view. Take *map* as an example, the function is bijective in terms of the list structure. As a result, a *put* function can share the recursion structure of the *get*, mapping whatever structural changes from the view back to the source. This becomes harder with the loss of structure information in the view. Take *append* as an example. The boundary between the first source list, which *get* recurs on, and the second source list is gone in the view. As a result, if a *put* function is to share the recursive structure of the *get*, the backward execution will always try to replenish the first source list first before leaving the remaining view elements as the second source list[7]. This is what *appendB* does. Any divergence from this behavior will require a different recursive structure for *put*, which drastically increases the search space as it loses the guidance of the *get*-based sketch.

We thus expect that the performance of Synbit varies according to the difficulty classes. For Class-1 problems, synthesis is likely to be successful for any given input/output examples (thus handling any structural changes); for Class-2 problems, synthesis is likely to be successful for some given input/output examples; and for Class-3 problems, synthesis is only possible for input/output examples that are free from structural changes.

The benchmark programs and the synthesis results are summarized in Table 1, which should be read together with Table 2 where the input/output examples used for the experiments are shown (which can also be used as a reference for the forward execution behaviors of the input functions). Also, we used the following auxiliary functions: equality over natural numbers for *lengthTail*, *reverse* and *appendBc*, and *length* in addition for the latter two. The definitions of all the functions

[6] https://hackage.haskell.org/package/criterion
[7] unless we know that the second list is fixed as in *appendBc*

Table 1. The results of experiments for categorized examples

Problem	Recursion on	Class	Result	Time (s)
double	Nat	1	Yes	0.050
uncurryReplicate	Nat	1	Yes	0.050
mapNot	List	1	Yes	0.052
mapReplicate	Nat and List	1	Yes	0.13
snoc	List	1	Yes	0.15
length	List	1	Yes	0.040
lengthTail	List (tail recursive)	1	Yes	0.22
reverse	List (tail recursive)	1	Yes	1.3
mapFst	List	1	Yes	0.016
add	Nat	2	Yes	0.045
append	List	2	Yes	0.034
appendBc	List	2	Yes	5.6
professor	List	2	Yes	0.023
lines	List	2	Timeout	-
lookup	List	3	Timeout	-

listed and the full synthesis results can be found in the artifact [8] or the repository[9] together with the implementation.

Class 1. As we can see, Synbit handles programs in this class with ease. An interesting case is *reverse*. On the conceptual level, the function is embarrassingly bijective and should be straightforward to invert. However, in practice the story is much more complicated, especially for the linear-time accumulative list reversal (the naive non-accumulative implementation has quadratic complexity in a functional language). It is well known in the program inversion literature (Matsuda et al. 2012, Nishida and Vidal 2011) that tail recursive functions (which are often needed for accumulation) are challenging to handle due to overlapping branch bodies. The *reverse* definition we use in the benchmark includes a small fix: it takes an additional parameter that represents the length of the list in the accumulation parameter. It is sufficient to guarantee the success of Synbit.

Class 2. As we can see, Synbit also performs well for this class. But as explained above, the success is conditional on the input/output examples that the *put* is required to satisfy. Take *append* as an example, if the following example is included, which demands the second list being filled before the original first list is fully reconstructed, the synthesis will fail, as a solution must have a different recursion structure from that of the sketch.

Original Source	(Original View)	Updated View	Updated Source
([1, 2, 3, 4], [5])	[1, 2, 3, 4, 5]	[6, 2]	([6], [2])

An interesting case is *lines*, which splits a string by `'\n'` to produce a list of strings. The synthesis becomes a lot harder when the examples (as seen in Table 2) require the preservation of the existence of the newline in the last position. This combined with structural changes to the view list cannot be captured by the recursion structure of the sketch, which explains the failure.

Class 3. Functions such as *lookup* completely lose the source structures. Consequently, Synbit will not be able to handle any example of structural changes. In the case of *lookup*, a structural change means that the view value is changed to another value associated to a different key in the source (as seen in Table 2). Just for demonstration, if only non-structural changes are considered,

[8] https://doi.org/10.5281/zenodo.5494504
[9] https://github.com/masaomi-yamaguchi/synbit

Table 2. Input/output examples: for readability, we shall write n for S^n Z (integer constants are also used in *snoc*, *reverse*, *mapFst* and *append*), and $st_n/pr_n/pr'_n$ for Student "st$n$"/Professor "pr$n$"/Professor "pr$n$'".

Program	Original Source	(Original View)	Updated View	Updated Source
double	1 5	2 10	6 4	3 2
uncurryReplicate	('a', 2) (True, 3)	"aa" [True, True, True]	"bbb", [False,False]	('b', 3) (False, 2)
mapNot	[True, False]	[False, True]	[True, True, False]	[False, False, True]
mapReplicate	[('b', 2)] [('a', 3), ('b', 1), ('c', 2)]	["bb"] ["aaa", "b", "cc"]	["aaa", "b", "cc"] ["bb"]	[('a', 3), ('b', 1), ('c', 2)] [('b', 2)]
snoc	([1, 2, 3], 4) ([1, 2, 3], 4)	[1, 2, 3, 4] [1, 2, 3, 4]	[1, 2, 3] [1, 2, 3, 4, 5, 6]	([1, 2], 3) ([1, 2, 3, 4, 5], 6)
length/*lengthTail*	[1, 2] [2, 0]	2 2	4 1	[1, 2, 0, 0] [2]
reverse	[True, True] [1, 2, 3, 4]	[True, True] [4, 3, 2, 1]	[False, True, True] [6, 5]	[True, True, False] [5, 6]
mapFst	[(1, 'a'), (2, 'b'), (3, 'c')] [(2, 'b'), (3, 'c')]	[1, 2, 3] [2, 3]	[1, 3] [0, 1, 2, 3]	[(1, 'a'), (3, 'b')] [(0, 'b'), (1, 'c'), (2, 'a'), (3, 'a')]
add	(2, 3) (2, 3)	5 5	7 1	(2, 5) (1, 0)
append	([1, 2, 3, 4], [5]) ([1, 2, 3, 4], [5])	[1, 2, 3, 4, 5] [1, 2, 3, 4, 5]	[6, 2] [1, 2, 3, 4, 5, 6]	([6, 2], []) ([1, 2, 3, 4], [5, 6])
appendBc	"apple" "apple"	"apple;;" "apple;;"	"pineapple;;" "plum;;"	"pineapple" "plum"
professor	$[st_1, st_2, pr_1, st_3, pr_2]$ $[st_1, st_2, pr_1, st_3, pr_2]$	$[pr_1, pr_2]$ $[pr_1, pr_2]$	$[pr'_1, pr'_2, pr'_3]$ $[pr'_1]$	$[st_1, st_2, pr'_1, st_3, pr'_2, pr'_3]$ $[st_1, st_2, pr'_1, st_3]$
lines	"aa\nbb\ncc" "aa" "aa\n"	[”aa”, ”bb”, ”cc”] [”aa”] [”aa”]	["aa", "bb"] ["aa", "bb"] ["aa", "bb"]	"aa\nbb" "aa\nbb" "aa\nbb\n"
lookup	([(1, 10), (2, 200), (3, 33)], 2) ([(1, 10), (2, 200), (3, 33)], 2)	200 200	10 33	([(1, 10), (2, 200), (3, 33)], 1) ([(1, 10), (2, 200), (3, 33)], 3)

as in the following example where the changed view does not switch to a different key, Synbit will be able to successfully generate a program.

Original Source	(Original View)	Updated View	Updated Source
([(1, 10), (2, 200), (3, 33)], 2)	200	10	([(1, 10), (2, 10), (3, 33)], 2)

However, this is not interesting as the strength of HOBiT lies in its ability to handle structural changes through branch switching.

4.2 Larger and More Involved Example

Next, we evaluate Synbit on some larger examples, which are closer to realistic use cases. In particular, we look at two types of transformations: XML queries and string parsing.

4.2.1 XML Transformations. We examined six queries from XML Query Use Cases[10] ("TREE" Use Case). Table 3 provides brief explanations for these queries and Figure 1 shows the skeleton of the XML document used as the original source for them. Such XML documents are represented in HOBiT by a rose-tree datatype. We ignored Document Type Definitions for simplicity—we could handle such constraints by fusing a partial identity function checking them to a *get* function. We also provided the constant "title" as an auxiliary component to our synthesis engine.

[10] https://www.w3.org/TR/xquery-use-cases

```
<book><title>Data on the Web</title>
<author>Serge Abiteboul</author><author>Peter Buneman</author><author>Dan Suciu</author>
<section id="intro" difficulty="easy">
<title>Introduction</title><p>. . . </p>
<section><title>Audience</title><p>. . . </p></section>
<section>
<title>Web Data and the Two Cultures</title>
<p>. . . </p>
<figure height="400" width="400">
<title>Traditional client/server architecture</title><image source="csarch.gif"/>
</figure>
<p>. . . </p>
</section>
</section>
<section id="syntax" difficulty="medium">. . . </section>
</book>
```

Fig. 1. An XML document used as an original source for Queries Q1 to Q6.

Table 3. Explanations of the examined XML queries: the descriptions are quoted from XML Query Use Case, where "Book1" refers the source XML.

Problem	Description (quoted)
Q1	"Prepare a (nested) table of contents for Book1, listing all the sections and their titles. Preserve the original attributes of each <section> element, if any."
Q2	"Prepare a (flat) figure list for Book1, listing all the figures and their titles. Preserve the original attributes of each <figure> element, if any."
Q3	"How many sections are in Book1, and how many figures?"
Q4	"How many top-level sections are in Book1?"
Q5	"Make a flat list of the section elements in Book1. In place of its original attributes, each section element should have two attributes, containing the title of the section and the number of figures immediately contained in the section."
Q6	"Make a nested list of the section elements in Book1, preserving their original attributes and hierarchy. Inside each section element, include the title of the section and an element that includes the number of figures immediately contained in the section."

Table 4 contains the results of this experiment. Column "Updates" indicates the updates of the source query triggered by the given input/output examples, whereas columns "$\mathrm{LOC}_{\mathrm{in}}$" and "$\mathrm{LOC}_{\mathrm{syn}}$" denote the number of lines of code in the original and the synthesized query, respectively. The results show that Synbit can synthesize fairly large HOBiT programs. In particular, the number of AST nodes synthesised ranges from 73 (for Q4) to 471 (for Q5), corresponding to 17 lines of code for Q4 and 80 for Q5. (The programs are too large to be displayed in this document. We refer interested readers to our extended report (Yamaguchi et al. 2021) for a full XML example.) The reason Q6 takes significantly more time than the rest is that it assumes that each section has a title element. Consequently, when handling insertion of sections, the generated reconciliation function needs to construct a section with a title.

4.2.2 Lexer and Parser. We also examined a simple recursive decent (specifically, LL(1)) lexer and parser. The lexer takes in strings (i.e., $\{(,), \mathsf{S}, \mathsf{Z}, +\}^*$) and returns a sequence of tokens represented

Table 4. The results of experiments for XML examples.

Problem	LOC_{in}	AST_{in}	Updates in I/O examples	Time (s)	LOC_{syn}	AST_{syn}
Q1	11	136	Add attribute(s) Remove section(s) Change title(s) and attribute value(s)	0.35	42	319
Q2	23	199	Remove figure(s) Change title(s) Change attribute value(s) Add Attribute	0.97	69	406
Q3	20	191	Decrease figure count Increase section count Decrease section count	0.43	63	339
Q4	9	97	Increase section count Decrease section count	0.14	26	170
Q5	35	342	Decrease figure count Increase figure count Remove section(s) Change title(s)	1.2	115	813
Q6	31	236	Increase figure count(s) Decrease figure count(s) Add section(s) with title and figure count Change title(s)	10	90	530

Table 5. Experimental results for the lexer and parser.

	LOC_{in}	AST_{in}	Updates	Time (s)	LOC_{syn}	AST_{syn}
Lexer	15	88	Change on natural numbers remove/insert tokens	0.094	69	265
Parser	10	45	Replacement of whole AST and back	0.55	29	121

by the following datatype.

```
data Token = TNum Nat | LPar | RPar | Plus
```

Note that natural numbers such as `S(S(Z))` are processed in this step. Then, the parser takes in the output of the lexer, i.e., a sequence of the tokens above, and returns an abstract syntax tree, according to the following grammar.

$$s ::= n \mid (s)+(s)$$

The lexer and parser considered here are injective, which is uncommon in practice. Typically, a lexer loses information about white spaces (layouting) and comments, and a parser may remove syntactic sugars and redundant parentheses[11]. Sometimes, though, such lost information is attached to the abstract syntax trees, making the parsing process injective (de Jonge and Visser 2011, Kort and Lämmel 2003, Pombrio and Krishnamurthi 2014).

Table 5 summarizes the experimental results. In Figure 2, we provide the parser generated by Synbit as it is the more intricate of the two. Notice that the LPAR case in *go* requires quite an involved reconciliation function.

[11]A notable exception is the parser for GHC/Haskell, which keeps syntactic sugars and parentheses for better error messaging.

```
pExp :: B[Token] → BExp
pExp ts = let (e, [ ]) = go ts in e
go :: B[Token] → B(Exp, [Token])
go ts = case ts of
  TNum n : r → (ENum n , r)
               with λv. case v of {(ENum _, _) → True; _ → False}
               by   λs.λv. case v of {(ENum a, _) → TNum a : s}
  LPAR : r1  → let (e1, RPar : Plus : LPar : r2) = go r1 in
               let (e2, RPar : Plus : LPar : r3) = go r2 in
                 (EAdd e1 e2 , r3)
               with λv. case v of {(EAdd _ _, _) → True; _ → False}
               by   λs.λv. case v of {(EAdd _ _, _) → LPar : TNum Z : RPar : Plus : LPar : TNum Z : RPar : s}
```

Fig. 2. Synthesized Bidirectional Parser

Table 6. Results of comparative experiments with Smyth: "No" means that Smyth reported failure in 10 min.

Problem	Class	Synbit	Smyth	Problem	Class	Synbit	Smyth
double	1	Yes	Yes	*mapFst*	1	**Yes**	No
uncurryReplicate	1	Yes	Yes	*add*	2	**Yes**	No
mapNot	1	Yes	Yes	*append*	2	**Yes**	No
mapReplicate	1	Yes	Yes	*appendBc*	2	**Yes**	No
snoc	1	**Yes**	No	*professor*	2	**Yes**	No
length	1	Yes	Yes	*lines*	2	Timeout	Timeout
lengthTail	1	Yes	Yes	*lookup*	3	Timeout	**Yes**
reverse	1	**Yes**	No				

4.3 Comparison with Smyth

A fair comparison with other synthesis systems is not always easy due to the different set-ups. For example, we cannot compare directly with Optician (Maina et al. 2018, Miltner et al. 2018, 2019), the state of the art lens synthesizer, as its inputs and outputs are too different from ours (see Section 5 for a non-experimental comparison).

Instead, we pick Smyth (Lubin et al. 2020), a state-of-the-art synthesis tool that synthesizes unidirectional programs from sketches and input/output examples—a set-up that is similar to ours. We provide to Smyth hand-written sketches of *put* in the form of "base case sketches" (Lubin et al. 2020), which are incomplete programs for which the step case branches are left as holes while the base case branches are pre-filled, and the same input/output examples as the experiments in Table 2. We omit the round-tripping requirement for Smyth and only check whether the tool is able to produce *put* functions that satisfy the input/output examples.

Table 6 shows the results of the comparison (with more details in the artifact [12] or the repository[13]). Synbit successfully synthesized 13 out of 15 cases, whereas Smyth succeeded only in 7 cases. We believe that the main reason for the difference is the required *put* functions tend to be quite complex, usually more so than their corresponding *get*. It is worth noting that Smyth succeeded for *lookup*, where Synbit failed. For this particular case, a *put* program that conforms to the input/output example is represented by the key-value-flipped version of *get*, which was ruled out in Synbit by a sketch.

[12] https://doi.org/10.5281/zenodo.5494504
[13] https://github.com/masaomi-yamaguchi/synbit

5 RELATED WORK

Optician. Optician (Maina et al. 2018, Miltner et al. 2018, 2019) is the state-of-the-art framework for synthesizing lenses (Bohannon et al. 2008, Foster et al. 2007, 2008, Hofmann et al. 2011). Both their framework and ours implicitly guarantee the round-tripping properties by using bidirectional programming languages (lenses/HOBiT) as targets. However, a direct comparison of performance is difficult due to the very different set-ups. Their target lenses are specialized for string transformations, while HOBiT considers general datatypes. And correspondingly, the core of their input specification is regular expressions describing data formats, while that of ours is standard functional programs serving as sketches. Due to such differences in set-ups, even though we could translate a specification for Optician (regular expressions and input/output examples) to one for Synbit (a *get* program and input/output examples), such a translation would involve many arbitrary choices (especially the choice of a *get* for Synbit) that affect synthesis, effectively ruling out a meaningful comparison (see Appendix A.1 for an illustrative example on the difficulty).

Despite the very different approaches, it is interesting to observe a common design principle shared by both: to leverage the strengths of the underlying bidirectional languages. Optician's regular-expression-based specification matches perfectly with the simplicity of the lens languages and their close connection to advanced types, while Synbit takes full advantage of HOBiT's alignment to conventional functional programming. On a more technical note, Optician (Miltner et al. 2019) is able to prioritize generated programs by quantitative information flow. It is not clear how this may be used in Synbit as the computation of the quantitative information flow will be difficult for a language with arbitrary recursion.

Other synthesis efforts for bidirectional programming. In a vision paper, Voigtländer (2012) suggests some directions of synthesizing bidirectional programs from *get* programs by leveraging the round-tripping properties. Specifically, he suggests using the round-tripping properties to generate input/output examples for synthesis. Using **Acceptability**, if one can generate s in some ways (assuming the totality of *get*), then examples of backward behavior may arise from $put\ (s, get\ s) = s$. But a naive application of this without considering **Consistency** may result in incorrect *put* behavior such as $put\ (s, v) = s$. To remedy the situation, Voigtländer suggests restricting *put* to use the second argument v; i.e., the argument must be relevant in the sense of relevant typing. Voigtländer also suggests using **Consistency** to restrict the form of *put* to satisfy $put\ (s, v) \in get^{-1}(v)$, which is indeed effective for simple *get*s such as $get = head$ (in this case the right-hand side must have the form of $v : \_$). However, synthesis in this direction does not guarantee correctness with respect to round-tripping, and an additional verification process will then be needed.

The PINS framework (Srivastava et al. 2011) applies path-based synthesis to program inversion, a program transformation that derives the inverse of an (injective) program. The path-based synthesis was able to derive inverses for involved programs such as LZ77 and LZW compression which the other existing inversion methods at the present time cannot handle. However, since it focuses only on a finite number of paths, the system does not guarantee correctness: the resulting programs may not always be inverses. PINS also uses sketches and component functions given by users.

Program inversion. Program inversion is a technique related to bidirectional programming; but there are important differences. In program inversion, input programs are expected to be injective and thus serve as complete specifications, which is not the case in bidirectional programming. As a result, in Synbit input/output examples are used to further specify the required backward behavior. Despite the differences, program inversion and bidirectional programming do share some common techniques. For example, using postconditions (as exit conditions in HOBiT) to determine control flows (especially branches) in inverses is a very common approach in the literature (Glück

and Kawabe 2005, Gries 1981, Korf 1981, Lutz 1986, Matsuda et al. 2010, Yokoyama et al. 2008, 2011). A more interesting connection is the concept of *partial* inversion (Nishida et al. 2005), which uses binding-time analysis before inversion so that the inverses can use static data as inputs as well. Types in HOBiT can be seen as binding time where non-**B**-types are seen as static, and our type-directed sketch generation (Section 3.3), with the lazy nondeterministic generation, can be viewed as a type-based binding-time analysis (Gomard and Jones 1991). The idea of partial inversion is further extended so that the return values of inverses are treated as "static inputs" as well (Almendros-Jiménez and Vidal 2006), and the *pin operator* (Matsuda and Wang 2020) is proposed to capture such a behavior in an invertible language. However, the utility of the operator in bidirectional programming rather than invertible programming is still under exploration, and thus our current synthesis method does not include it.

Bidirectionalization. Bidirectionalization is a program transformation that derives a bidirectional transformation from a unidirectional transformation. In a sense, this can be seen as a simple type of synthesis. Matsuda et al. (2007), based on the constant-complement view updating (Bancilhon and Spyratos 1981), analyze injectivity (information-loss) of a program and then derive a complement by gathering lost information to obtain a bidirectional version. This method requires a strong restriction on input programs for effective analysis: they must be affine (no variables can be used more than once) and treeless (Wadler 1990) (only variables can be arguments of functions) so that the injectivity analysis becomes exact. Voigtländer (2009) makes use of parametricity (Reynolds 1983, Wadler 1989) to interpret polymorphic functions as bidirectional transformations. The technique is restricted to polymorphic functions. And probably more importantly, it can only handle non-structural updates—the equivalent of HOBiT without the ability of branch switching. Several extensions of the idea have been proposed (Matsuda and Wang 2015a, 2018a, Voigtländer et al. 2013). But in general, bidirectionalization is far less expressive than the state-of-the-art synthesis frameworks such as Optician/lenses and Synbit/HOBiT.

General program synthesis. A popular direction in program synthesis that inspired our work is program sketching, where programmers express their insights about a program by writing sketches, i.e., partial programs encoding the structure of a solution while leaving its low-level details unspecified in the form of holes (Solar-Lezama 2009). As opposed to our technique, Solar-Lezama (2009) can only be applied to integer benchmarks and does not support other data types such as lists or trees. Also, it mostly focuses on properties, rather than examples, by relying on Counterexample Guided Inductive Synthesis (CEGIS) (Solar-Lezama et al. 2008), where a candidate solution is iteratively refined based on counterexamples provided by a verification technique. There is actually a large body of works based on the CEGIS architecture (Abate et al. 2018, Jha et al. 2010, Kneuss et al. 2013). Usually, such approaches expect formal specifications describing the behavior of the target program, which are often unavailable, difficult to write, or expensive to check against using automated verification techniques. Conversely, our specification consists of the unidirectional program and input/output examples, which we believe is intuitive and easy to use, without requiring prior understanding of logic.

The original work on program sketching has inspired a multitude of follow-up directions. Some of the most related to our work are Feser et al. (2015), Katayama (2005), Lubin et al. (2020), Osera and Zdancewic (2015), which, similarly to our technique, are type-directed and guided by input/output examples. As opposed to these approaches, we exploit information about the unidirectional program in order to prune the search space for the bidirectional correspondent. As shown in our experimental evaluation, simply applying synthesis techniques designed for unidirectional code is not effective.

Another direction that inspired us is that of component-based synthesis (Feng et al. 2017, Jha et al. 2010), where the target program is generated by composing components from a library. Similarly

to these approaches, we use a given library of components as the building blocks of our program generation approach.

Equivalence reduction. Program synthesis techniques make use of equivalence reduction in order to reduce the number of equivalent programs that get explored. For example, Albarghouthi et al. (2013) prune the search space using observational equivalence with respect to a set of input/output examples, i.e., two programs are considered to be in the same equivalence class if, for all given inputs in the set of input/output examples, they produce the same outputs. Alternatively, Smith and Albarghouthi (2019) generate only programs in a specific normal form, where term rewriting is used to transform a program into its normal form. In (Koukoutos et al. 2016), Koukoutos et al. make use of attribute grammars to only produce certain types of expressions in their normal form, thus skipping other expressions that are syntactically different, yet semantically equivalent. In our work, we found that the lightweight heuristics described in Section 3.6 worked well. However, we do plan on exploring some of the equivalence reduction techniques discussed here as future work.

6 CONCLUSION

We proposed a synthesis method for bidirectional transformations, whose novelty lies in the use of *get* programs as sketches. We described the idea in detail and implemented it in a prototype system Synbit, where lazy nondeterministic generation has played an important role. Through the experiments, we demonstrated the effectiveness of the proposed method and clarified its limitations.

A future direction is to make use of program analysis and verification techniques in the synthesis of exit conditions. This would enable us to guarantee stronger soundness as discussed in Section 3.7. Another future direction is to extend the target language (HOBiT) based on our experience in order to synthesize more bidirectional transformations.

ACKNOWLEDGMENTS

We thank Eijiro Sumii and Oleg Kiselyov for their helpful and instructive comments on an earlier stage of this research, and Hiroshi Unno for fruitful discussions on future directions. This work was partially supported by JSPS KAKENHI Grant Numbers 15H02681, 19K11892 and 20H04161, JSPS Bilateral Program, Grant Number JPJSBP120199913, the Kayamori Foundation of Informational Science Advancement, EPSRC Grant *EXHIBIT: Expressive High-Level Languages for Bidirectional Transformations* (EP/T008911/1), Royal Society Grant *Bidirectional Compiler for Software Evolution* (IES\R3\170104), and Royal Society University Research Fellowship *On Advancing Inductive Program Synthesys* (UF160079).

A APPENDIX

A.1 More Discussion on the Difficulty of Side-by-Side Comparison with Optician

Due to the very different set-ups, a side-by-side comparison of Synbit and Optician is problematic. The arbitrary choices required to bridge the gap make a fair comparison out of reach. We illustrate this problem with an example (`extr-fname.boom` in Fig. 3) taken from the artifact associated with the Optician papers (Miltner et al. 2018, 2019).

The specification describes the task of separating a path into a file and a directory path. As one can see, most of the code is devoted to specifying the input and output formats (*NONEMPTYDIRECTORY* and *FILEANDFOLDER*). The input/output examples are specified by the **using** clause: **createrex** provides an example of how a source is related to a view. Note that this specification targets the synthesis of bijective transformations; so the backward behavior does not require the original source.

```
let LOWERCASE : regexp = "a" | "b" | ... (* omitted *) ··· | "z"
let UPPERCASE : regexp = "A" | "B" | ... (* omitted *) ··· | "Z"

let LOCALFOLDER : regexp =
    (LOWERCASE | UPPERCASE | "_" | "." | "-")
  . (LOWERCASE | UPPERCASE | "_" | "." | "-")*
let DIRECTORY : regexp = ("/" | "") . (LOCALFOLDER . "/")*
let NONEMPTYDIRECTORY : regexp =
    ("/" | "") . LOCALFOLDER . ("/" . LOCALFOLDER)*
let FILEANDFOLDER : regexp =
    "file: " . LOCALFOLDER . "\nfolder: " . DIRECTORY
let extract_file : (lens in NONEMPTYDIRECTORY ⇔ FILEANDFOLDER) =
  synth NONEMPTYDIRECTORY ⇔ FILEANDFOLDER
  using {
    createrex("/Users/amiltner/lens/tests/flashfill/extract-filename.txt",
              "file: extract-filename.txt\nfolder: /Users/amiltner/lens/tests/flashfill/"),
    createrex("tests/flashfill/extract-filename.txt",
              "file: extract-filename.txt\nfolder: tests/flashfill/")
  }
```

Fig. 3. `extr-fname.boom` for bijective-lens synthesis (excerpt)

Let us consider how we can encode this specification to be used by Synbit. As a first step, we need to decide the types for inputs and outputs. One candidate is using strings (lists of characters in HOBiT). In such a case, it is natural to divide the task into three subtasks: (1) parsing (of type String $\to S$), (2) core transformation (of type $S \to T$), and (3) printing (of type $T \to$ String), such that the interesting computation is done in the middle. For the comparison to Optician, it makes sense to only consider the core transformation; parsing and printing are coupled with lens combinators used in Optician and are not synthesized separately from the core transformation.

We then need to decide the domain (S) and range (T) of the core transformation. One option is to use S = (NonEmpty String, Bool) and T = (String, Bool, [String]), where:

```
type NonEmpty a = (a, [a])        -- head-biased non-empty lists
```

Another option is to use datatypes that mirror the structure of regular expressions, such as:

```
data LC = LA | LB | ··· | LZ
data UC = UA | UB | ··· | UZ
data C = Lower LC | Upper UC | UnderScore | Dot | Hyphen
type LocalFolder = (C, [C])
type Directory = (Bool, [LocalFolder])
type NonEmptyDirectory = (Bool, LocalFolder, [LocalFolder])
type FileAndFolder = (LocalFolder, Directory)
```

In this particular case, the choice between the two does not affect the core transformation part much; in both cases, it essentially performs a transformation from head-biased nonempty lists to last-biased ones, with some arrangement of products. So, one can think that the essential part of this transformation is a function of type $headBiased2LastBiased :: (A, [A]) \to ([A], A)$ for some concrete type A. Note that abstracting the concrete type A by a type variable a here gives us the information that the components of the lists are not touched by the transformation.

The above set-up may sound reasonable but actually omits important internal details. Optician internally tries to expand $a*$ into either $aa*|\varepsilon$ or $a*a|\varepsilon$ nondeterministically (Miltner et al. 2019), which eventually transforms $aa*$ (head-biased non-empty lists) into $a(a*a|\varepsilon) = aa*a|a$ (one-step

expansions of last-biased nonempty lists). The core transformation involves no structural transformations after this expansion. However, this expansion of Kleene star conflicts with Synbit where the input and output types have to be fixed beforehand. Optician dynamically searches for a suitable-for-synthesis regular expression among equivalent ones mainly by converting them to "sum-of-product" forms and then by applying the expansion above (Miltner et al. 2019).

Trying to give a concrete definition of the transformation is even more problematic, with semantically equivalent definitions having very different effects on synthesis. For example, if we define *headBiased2LastBiased* :: $(A, [A]) \to ([A], A)$ as the following:

$$\begin{aligned}
&\mathit{headBiased2LastBiased}\ (a, as) = \mathit{initlast}\ a\ as \\
&\mathit{initlast}\ a\ [\,] \quad\ = ([\,], a) \\
&\mathit{initlast}\ a\ (b : bs) = \mathbf{let}\ (i, l) = \mathit{initlast}\ b\ bs\ \mathbf{in}\ (a : i, l)
\end{aligned}$$

Synbit has no problem in synthesizing a bidirectional version of it. On the other hand, the following equivalent definition does not work well.

$$\begin{aligned}
&\mathit{headBiased2LastBiased}\ (a, as) = (\mathit{init}\ a\ as, \mathit{last}\ a\ as) \\
&\mathit{init}\ a\ [\,] \quad\ = [\,] \\
&\mathit{init}\ a\ (b : bs) = a : \mathit{init}\ b\ bs \\
&\mathit{last}\ a\ [\,] \quad\ = a \\
&\mathit{last}\ a\ (b : bs) = \mathit{last}\ b\ bs
\end{aligned}$$

The reason for this is that the bijective transformation is separated into non-injective components *init* and *last*. Non-injectivity is usually not a problem as Synbit is designed to handle them with *put*. But in this case, the information that the non-injective functions are combined to form a bijection is lost in the separation, which restricts the updates that the backward function may handle. Synbit will (correctly) insists that the input data discarded by *init*/*last* cannot be changed in the backward execution (otherwise, the round-tripping properties will be (locally) violated), which in this case results in a useless bidirectional program that rejects all changes (and of course the synthesis fails at this point as the input/output examples cannot be satisfied).

In a similar manner, the opposite direction of encoding Synbit examples in Optician is also problematic. A lot of cases will simply fail to translate and for the rest particular ways of encoding is required for Optician to work well. Due to this, a side-by-side comparison of the two systems will be forced and unlikely to produce meaningful results.

It is apparent that Optician and Synbit occupy very different parts of the synthesis design space. This difference is driven by the differences in the underlying languages they target: lenses vs HOBiT. Lenses are tricky to program with but the language itself is very simple; it therefore makes sense to have a separate specification system that is removed from the target implementation. In contrast, HOBiT focuses more on programmability and the specification system may naturally take advantage of the fact. In a sense, lenses may be considered to benefit more from synthesis, as it relieves the need to program directly in them. On the other hand, Synbit demonstrates the impact of the language design: it not only improves programability, but also enables effective synthesis methods.

REFERENCES

Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample Guided Inductive Synthesis Modulo Theories. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10981)*, Hana Chockler and Georg Weissenbacher (Eds.). Springer, 270–288. https://doi.org/10.1007/978-3-319-96145-3_15

Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 934–950. https://doi.org/10.1007/978-3-642-39799-8_67

Jesús Manuel Almendros-Jiménez and Germán Vidal. 2006. Automatic Partial Inversion of Inductively Sequential Functions. In *Implementation and Application of Functional Languages, 18th International Symp osium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4449)*, Zoltán Horváth, Viktória Zsók, and Andrew Butterfield (Eds.). Springer, 253–270. https://doi.org/10.1007/978-3-540-74130-5_15

François Bancilhon and Nicolas Spyratos. 1981. Update Semantics of Relational Views. *ACM Trans. Database Syst.* 6, 4 (1981), 557–575. https://doi.org/10.1145/319628.319634

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 407–419. https://doi.org/10.1145/1328438.1328487

Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. 2020. Code-level model checking in the software development workflow. In *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 11–20. https://doi.org/10.1145/3377813.3381347

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (2001), 555–604. https://doi.org/10.1145/382780.382785

Maartje de Jonge and Eelco Visser. 2011. An Algorithm for Layout Preservation in Refactoring Transformations. In *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6940)*, Anthony M. Sloane and Uwe Aßmann (Eds.). Springer, 40–59. https://doi.org/10.1007/978-3-642-28830-2_3

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 599–612. http://dl.acm.org/citation.cfm?id=3009851

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 229–239. https://doi.org/10.1145/2737924.2737977

Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. 2011. Purely functional lazy nondeterministic programming. *J. Funct. Program.* 21, 4-5 (2011), 413–465. https://doi.org/10.1017/S0956796811000189

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007). https://doi.org/10.1145/1232420.1232424

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. 2008. Quotient lenses. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 383–396. https://doi.org/10.1145/1411204.1411257

Robert Glück and Masahiko Kawabe. 2005. Revisiting an automatic program inverter for Lisp. *SIGPLAN Notices* 40, 5 (2005), 8–17.

Carsten K. Gomard and Neil D. Jones. 1991. A Partial Evaluator for the Untyped lambda-Calculus. *J. Funct. Program.* 1, 1 (1991), 21–69. https://doi.org/10.1017/S0956796800000058

David Gries. 1981. *The Science of Programming.* Springer, Heidelberg, Chapter 21 Inverting Programs.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 317–330. https://doi.org/10.1145/1926385.1926423

Stephen J. Hegner. 1990. Foundations of Canonical Update Support for Closed Database Views. In *ICDT (Lecture Notes in Computer Science, Vol. 470)*, Serge Abiteboul and Paris C. Kanellakis (Eds.). Springer, 422–436. https://doi.org/10.1007/3-540-53507-1_93

Martin Hofmann, Benjamin C. Pierce, and Daniel Wagner. 2011. Symmetric lenses. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 371–384. https://doi.org/10.1145/1926385.1926428

Zhenjiang Hu and Hsiang-Shang Ko. 2016. Principles and Practice of Bidirectional Programming in BiGUL. In *Bidirectional Transformations - International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures (Lecture Notes in Computer*

Science, Vol. 9715), Jeremy Gibbons and Perdita Stevens (Eds.). Springer, 100–150. https://doi.org/10.1007/978-3-319-79108-1_4

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel (Eds.). ACM, 215–224. https://doi.org/10.1145/1806799.1806833

Susumu Katayama. 2005. Systematic search for lambda expressions. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005 (Trends in Functional Programming, Vol. 6)*, Marko C. J. D. van Eekelen (Ed.). Intellect, 111–126.

Oleg Kiselyov and Simon J. Thompson (Eds.). 2012. *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*. ACM. http://dl.acm.org/citation.cfm?id=2103746

Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes (Eds.). ACM, 407–426. https://doi.org/10.1145/2509136.2509555

Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: a formally verified core language for putback-based bidirectional programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 61–72. https://doi.org/10.1145/2847538.2847544

Richard E. Korf. 1981. Inversion of Applicative Programs. In *IJCAI*, Patrick J. Hayes (Ed.). William Kaufmann, 1007–1009.

Jan Kort and Ralf Lämmel. 2003. Parse-Tree Annotations Meet Re-Engineering Concerns. In *3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003), 26-27 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 161. https://doi.org/10.1109/SCAM.2003.1238042

Manos Koukoutos, Etienne Kneuss, and Viktor Kuncak. 2016. An Update on Deductive Synthesis and Repair in the Leon Tool. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016 (EPTCS, Vol. 229)*, Ruzica Piskac and Rayna Dimitrova (Eds.). 100–111. https://doi.org/10.4204/EPTCS.229.9

Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program sketching with live bidirectional evaluation. *Proc. ACM Program. Lang.* 4, ICFP (2020), 109:1–109:29. https://doi.org/10.1145/3408991

Christopher Lutz. 1986. Janus: a time-reversible language. (1986). *Letter to R. Landauer*. Available on: http://tetsuo.jp/ref/janus.pdf.

Solomon Maina, Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing quotient lenses. *Proc. ACM Program. Lang.* 2, ICFP (2018), 80:1–80:29. https://doi.org/10.1145/3236775

Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. 2007. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 47–58. https://doi.org/10.1145/1291151.1291162

Kazutaka Matsuda, Kazuhiro Inaba, and Keisuke Nakano. 2012. Polynomial-time inverse computation for accumulative functions with multiple data traversals, See (Kiselyov and Thompson 2012), 5–14. https://doi.org/10.1145/2103746.2103752

Kazutaka Matsuda, Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. 2010. A Grammar-Based Approach to Invertible Programs. In *ESOP (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 448–467.

Kazutaka Matsuda and Meng Wang. 2015a. Applicative bidirectional programming with lenses. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, Kathleen Fisher and John H. Reppy (Eds.). ACM, 62–74. https://doi.org/10.1145/2784731.2784750

Kazutaka Matsuda and Meng Wang. 2015b. "Bidirectionalization for free" for monomorphic transformations. *Sci. Comput. Program.* 111 (2015), 79–109. https://doi.org/10.1016/j.scico.2014.07.008

Kazutaka Matsuda and Meng Wang. 2018a. Applicative bidirectional programming: Mixing lenses and semantic bidirectionalization. *J. Funct. Program.* 28 (2018), e15. https://doi.org/10.1017/S0956796818000096

Kazutaka Matsuda and Meng Wang. 2018b. HOBiT: Programming Lenses Without Using Lens Combinators. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 31–59. https://doi.org/10.1007/978-3-319-89884-1_2

Kazutaka Matsuda and Meng Wang. 2020. Sparcl: a language for partially-invertible computation. *Proc. ACM Program. Lang.* 4, ICFP (2020), 118:1–118:31. https://doi.org/10.1145/3409000

Anders Miltner, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2018. Synthesizing bijective lenses. *Proc. ACM Program. Lang.* 2, POPL (2018), 1:1–1:30. https://doi.org/10.1145/3158089

Anders Miltner, Solomon Maina, Kathleen Fisher, Benjamin C. Pierce, David Walker, and Steve Zdancewic. 2019. Synthesizing symmetric lenses. *Proc. ACM Program. Lang.* 3, ICFP (2019), 95:1–95:28. https://doi.org/10.1145/3341699

Naoki Nishida, Masahiko Sakai, and Toshiki Sakabe. 2005. Partial Inversion of Constructor Term Rewriting Systems. In *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3467)*, Jürgen Giesl (Ed.). Springer, 264–278. https://doi.org/10.1007/978-3-540-32033-3_20

Naoki Nishida and Germán Vidal. 2011. Program Inversion for Tail Recursive Functions. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia (LIPIcs, Vol. 10)*, Manfred Schmidt-Schauß (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 283–298. https://doi.org/10.4230/LIPIcs.RTA.2011.283

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Stephen M. Blackburn (Eds.). ACM, 619–630. https://doi.org/10.1145/2737924.2738007

Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. 2014. Monadic combinators for "Putback" style bidirectional programming. In *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA*, Wei-Ngan Chin and Jurriaan Hage (Eds.). ACM, 39–50. https://doi.org/10.1145/2543728.2543737

Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: lifting evaluation sequences through syntactic sugar. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 361–371. https://doi.org/10.1145/2594291.2594319

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, R. E. A. Mason (Ed.). North-Holland/IFIP, 513–523.

Calvin Smith and Aws Albarghouthi. 2019. Program Synthesis with Equivalence Reduction. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, 24–47. https://doi.org/10.1007/978-3-030-11245-5_2

Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5904)*, Zhenjiang Hu (Ed.). Springer, 4–13. https://doi.org/10.1007/978-3-642-10672-9_3

Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. 2008. Sketching concurrent data structures. In *PLDI*. 136–148. https://doi.org/10.1145/1375581.1375599

Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. 2011. Path-based inductive synthesis for program inversion. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 492–503. https://doi.org/10.1145/1993498.1993557

Perdita Stevens. 2008. *A Landscape of Bidirectional Model Transformations.* Springer Berlin Heidelberg, Berlin, Heidelberg, 408–424. https://doi.org/10.1007/978-3-540-88643-3_10

Janis Voigtländer. 2009. Bidirectionalization for Free! (Pearl). In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/1480881.1480904

Janis Voigtländer. 2009. Bidirectionalization for free! (Pearl). In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, Zhong Shao and Benjamin C. Pierce (Eds.). ACM, 165–176. https://doi.org/10.1145/1480881.1480904

Janis Voigtländer. 2012. Ideas for connecting inductive program synthesis and bidirectionalization, See (Kiselyov and Thompson 2012), 39–42. https://doi.org/10.1145/2103746.2103757

Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. 2013. Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. *J. Funct. Program.* 23, 5 (2013), 515–551. https://doi.org/10.1017/S0956796813000130

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 347–359. https://doi.org/10.1145/99370.99404

Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (1990), 231–248. https://doi.org/10.1016/0304-3975(90)90147-A

Masaomi Yamaguchi, Kazutaka Matsuda, Cristina David, and Meng Wang. 2021. Synbit*: Synthesizing Bidirectional Programs using Unidirectional Sketches.* Technical Report.

Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. 2008. Principles of a reversible programming language. In *Proceedings of the 5th Conference on Computing Frontiers, 2008, Ischia, Italy, May 5-7, 2008*, Alex Ramírez, Gianfranco Bilardi, and Michael Gschwind (Eds.). ACM, 43–54. https://doi.org/10.1145/1366230.1366239

Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. 2011. Towards a Reversible Functional Language. In *RC (Lecture Notes in Computer Science, Vol. 7165)*, Alexis De Vos and Robert Wille (Eds.). Springer, 14–29. https://doi.org/10.1007/978-3-642-29517-1_2