Formal Methods in System Design (2022) 61:198–247
https://doi.org/10.1007/s10703-023-00436-9

SYNBIT: synthesizing bidirectional programs using unidirectional sketches

Masaomi Yamaguchi[1,2] · Kazutaka Matsuda[1] · Cristina David[3] · Meng Wang[3]

Received: 1 April 2022 / Accepted: 19 June 2023 / Published online: 29 January 2024

Abstract

We propose a technique for synthesizing bidirectional programs from the corresponding unidirectional code plus 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. This is an extended version of the paper "Synbit: Synthesizing Bidirectional Programs using Unidirectional Sketches", published at OOPSLA 2021. In addition to the OOPSLA'21 paper, this journal will contain additional formalization and detailed examples.

Keywords Program synthesis · Bidirectional transformation · Domain specific language · Programming by example · Functional language

✉ Kazutaka Matsuda
kztk@tohoku.ac.jp

Masaomi Yamaguchi
masaomi.yamaguchi.t4@dc.tohoku.ac.jp

Cristina David
cristina.david@bristol.ac.uk

Meng Wang
meng.wang@bristol.ac.uk

1 Graduate School of Information Sciences, Tohoku University, Sendai, Miyagi, Japan

2 Present Address: Fujitsu, Kawasaki, Japan

3 University of Bristol, Bristol, Avon BS8 1QU, UK

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 [1, 2], bidirectional model transformation [3], and so on).

As a response to this need, programming languages researchers started to design specialized programming languages for writing bidirectional transformations. In particular as pioneered by Pierce's group at Pennsylvania, a *bidirectional transformation* (BX), also known as a *lens* [4], 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*.

$$\begin{array}{lll r} \mathit{get}\ s = v & \text{implies} & \mathit{put}\ (s, v) = s & \text{(Acceptability)} \\ \mathit{put}\ (s, v) = s' & \text{implies} & \mathit{get}\ s' = v & \text{(Consistency)} \end{array}$$

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 [4], 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 Matsuda et al. [5], Bohannon et al. [6], Voigtländer [7], Pacheco et al. [8], Matsuda and Wang [9]). The state of the art has progressed a long way since. This includes a language HOBiT [9], which follows a line of research [5, 7, 10, 11] 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 [12], something that bidirectional programming research as a whole may benefit from.

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 familiar: users simply write normal unidirectional programs (together with 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 Sect. 5, 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 [5, 7] 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 [13] 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 Sect. 5.3 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 [14] and by seeding the program search with a sketch representing the program structure [15]. In this paper, we are interested in synthesizing general purpose programs and therefore we do not adopt the first strategy.

Contributions In this paper, we present an application of program synthesis to the area of bidirectional programming, by providing an automated technique for generating bidirectional transformations in the language HOBiT. Specifically,

- We design a system that takes unidirectional code and and concrete input/output examples as inputs (Sect. 3.2) and generate HOBiT code that execute bidirectionally.
- We optimize the synthesis by exploiting 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 (Sect. 3.3), which are then filled in a modular manner by separating the solving of dependent synthesis tasks (Sects. 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 (Sect. 5.1). 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 (Sect. 5). The prototype implementation of SYNBIT is available in the artifact[1] or the repository.[2]

2 Background: the HOBiT language

HOBiT [9] 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 that of 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 [9].

2.1 A simple example

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

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

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

A corresponding bidirectional program in HOBiT ($appendB :: \mathbf{B}[a] \rightarrow \mathbf{B}[a] \rightarrow \mathbf{B}[a]$) will include forward behaviour (*get*) just as *append*, and additionally suitable backward behavior (*put*). 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 [9]. As we will see in the rest of the section, bidirectional types can be mixed with normal unidirectional types to support flexible programming and greater expressiveness.

Bidirectional functions of type $\mathbf{B}\sigma \rightarrow \mathbf{B}\tau$ 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]
```

[1] https://doi.org/10.5281/zenodo.5494504

[2] https://github.com/masaomi-yamaguchi/synbit

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 $uncurryB :: (\mathbf{B}a \to \mathbf{B}b \to \mathbf{B}c) \to \mathbf{B}(a, b) \to \mathbf{B}c$ 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 [7], 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.

$$\begin{array}{l} appendB :: \mathbf{B}[a] \to \mathbf{B}[a] \to \mathbf{B}[a] \\ appendB\ xs\ ys = \underline{\mathbf{case}}\ xs\ \underline{\mathbf{of}}\ \begin{array}[t]{ll} [\,] & \to ys \\ a : x & \to a \mathbin{\underline{:}} appendB\ x\ ys \end{array} \end{array}$$

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 $\tau \to$ 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 a 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 [4, 9, 16–19]. When a branch 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 [9]. 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 (Sect. 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 :: $\mathbf{B}[a] \to [a] \to \mathbf{B}[a]$
appendBc xs ys = case xs of
    [ ]    → !ys
             with $\lambda v.\ length\ v == length\ ys$ by $\lambda\_.\lambda\_.\ [\,]$
    a : x → a : appendBc x ys
             with $\lambda v.\ length\ v \neq length\ ys$ by $\lambda\_.\lambda(v : \_).\ [v]$
```

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. Note that ! is an operator to make the program well-typed by *lifting* a constant to the bidirectional world.

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 HOBiT's original paper [9] 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 like 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. More details on the guarantees of our system are given later in Sect. 3.7.

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

$$\begin{array}{l}
\mathit{append} :: [\mathsf{Int}] \to [\mathsf{Int}] \to [\mathsf{Int}] \\
\mathit{append} = \lambda \mathit{xs}.\, \lambda \mathit{ys}.\, \mathbf{case}\ \mathit{xs}\ \mathbf{of}\ \{[\,] \to \mathit{ys};\ \ (a : x) \to a : \mathit{append}\ x\ \mathit{ys}\} \\
\texttt{:put}\ (\mathit{uncurryB}\ \mathit{appendB})\ ([1,2,3],[4,5])\ [6,2] = ([6,2],[\,])
\end{array}$$

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 Sect. 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. Note that the above *append* is specialized in Int. This is because polymorphism is not the main topic in our synthesis procedure. We focus on simple types in this section and will explain the treatment of polymorphism later in Sect. 4.

It is worth nothing that the input/output example above is in fact carefully chosen to trigger branch switching. One might notice that *append* [1, 2, 3] [4, 5] and *append* [6, 2] [ ] take different execution paths (in terms of branching structure). Hence, the input/output example tells SYNBIT to synthesize appropriate exit conditions and reconciliation functions for branching switching regarding such updates. In general, users are encouraged to provide examples that cover their intended updates, especially ones for which the given unidirectional program takes different execution paths to force SYNBIT to synthesize approaching exit conditions and reconciliation functions suitable for the branch switching.

For the given specification, SYNBIT produces the following result.

$$\begin{array}{l}
\mathit{appendB} :: \mathbf{B}[\mathsf{Int}] \to \mathbf{B}[\mathsf{Int}] \to \mathbf{B}[\mathsf{Int}] \\
\mathit{appendB} = \lambda \mathit{xs}.\, \lambda \mathit{ys}.\, \underline{\mathbf{case}}\ \mathit{xs}\ \underline{\mathbf{of}} \\
\quad [\,] \quad\ \ \to \mathit{ys} \\
\qquad\qquad\ \ \underline{\mathbf{with}}\ \lambda v.\, \mathbf{case}\ v\ \mathbf{of}\ \{x \to \mathsf{True};\ \_ \to \mathsf{False}\} \\
\qquad\qquad\ \ \underline{\mathbf{by}}\quad \lambda s.\lambda v.\, \mathbf{case}\ v\ \mathbf{of}\ \{x \to [\,]\} \\
\quad a : x \to a\ \underline{:}\ \mathit{appendB}\ x\ \mathit{ys} \\
\qquad\qquad\ \ \underline{\mathbf{with}}\ \lambda v.\, \mathbf{case}\ v\ \mathbf{of}\ \{z : \mathit{zs} \to \mathsf{True};\ \_ \to \mathsf{False}\} \\
\qquad\qquad\ \ \underline{\mathbf{by}}\quad \lambda s.\lambda v.\, \mathbf{case}\ v\ \mathbf{of}\ \{z : \mathit{zs} \to s\}
\end{array}$$

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

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

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 $\mathbf{B}[\mathsf{Int}] \to \mathbf{B}[\mathsf{Int}] \to \mathbf{B}[\mathsf{Int}]$, $\mathbf{B}[\mathsf{Int}] \to [\mathsf{Int}] \to \mathbf{B}[\mathsf{Int}]$, $[\mathbf{B}\mathsf{Int}] \to \mathbf{B}[\mathsf{Int}] \to \mathbf{B}[\mathsf{Int}]$, and so on. It is therefore crucial to narrow down the choices 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 Sect. 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 $\underline{\mathbf{case}}$, which expects exit conditions and reconciliation functions to be added (as holes (□) in the sketch).

$$
\begin{array}{l}
appendB = \lambda xs.\, \lambda ys.\, \underline{\mathbf{case}}\ xs\ \underline{\mathbf{of}} \\
\quad [\,] \quad\ \ \to ys \qquad\qquad\qquad\ \underline{\mathbf{with}}\ \square\ \underline{\mathbf{by}}\ \square \\
\quad a : x \to a\ \underline{:}\ appendB\ x\ ys \ \ \underline{\mathbf{with}}\ \square\ \underline{\mathbf{by}}\ \square
\end{array}
$$

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 $\underline{\mathbf{case}}$ 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.

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

In the spirit of component-based synthesis [20, 21], we generate expressions to fill holes in the sketch by composing components from a library that includes **case** and $\underline{\mathbf{case}}$ expressions, Bool constructors and operators, as well as list and tuple constructors. As we will explain in Sect. 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.

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

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 ($\square$) 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 $p \rightarrow e$, 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 should 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 $p \rightarrow e$, 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 Sect. 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 Sect. 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 Sect. 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.

```
appendB = λxs. λys. case xs of
    [ ]   → ys
            with λv. case v of {x → □₁; _ → False}
            by   λs.λv. case v of {x → □([ ])₃}
    a : x → a : appendB x ys
            with λv. case v of {z : zs → □₂; _ → False}
            by   λs.λv. case v of {z : zs → □(a : x)₄}
```

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 updated view [6, 2], and $\Box_2$ must evaluate to True in this context. Therefore, $\Box_2[6/z, [2]/zs, [6,2]/v] \equiv \mathsf{True}$.
(ii) *a* : *append x ys* (and therefore the v) has the value of the updated view [2], and $\Box_2$ must evaluate to True in this context. Therefore, $\Box_2[2/z, [\,]/zs, [2]/v] \equiv \mathsf{True}$.
(iii) *ys* (and therefore the v) has the value of [ ], and $\Box_1$ must evaluate to True in this context. Therefore, $\Box_1[[\,]/x, [\,]/v] \equiv \mathsf{True}$.

These constraints are useful in generating the exit conditions independently. As a matter of fact, in the case of *appendB* both $\Box_1$ and $\Box_2$ 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

$$
\begin{array}{lll}
\text{Expressions} & e ::= & x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{C}\ \overline{e} \mid \mathbf{case}\ \ e_0\ \mathbf{of}\ \{p_i \to e_i\}_i \\
 & \mid & \underline{\mathsf{C}}\ \overline{e} \mid \mathbf{case}\ \ e_0\ \mathbf{of}\ \{\underline{p_i} \to e_i\ \underline{\mathbf{with}}\ e'_i\ \underline{\mathbf{by}}\ e''_i\}_i \mid\ !e \\
\text{Patterns} & p ::= & x \mid \mathsf{C}\ \overline{p}
\end{array}
$$

Fig. 1 Syntax of (a part of) HOBiT: x ranges over variables, and C ranges over constructors

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 Sect. 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, where $f_i = e_i$ stands for a function/value definition of f_i by the value of e_i. The syntax of the expressions of HOBiT is shown in Fig. 1.
- $\Gamma = \{f_i : A_i\}_i$ 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 Sect. 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; for the k-th example, s_k is the original source, v_k is the updated view and s'_k is the updated source.

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 Sect. 3.1, the default library includes **case** and **case** expressions, Bool constructors and operators, as well as list and tuple constructors.

Example 1 (*append*) For the *appendB* example, the input is formally expressed as:

$$
\begin{aligned}
P_{\text{app}} &= \left\{ \begin{array}{l} \mathit{appendB} = \lambda xs.\ \lambda ys.\ \mathbf{case}\ xs\ \mathbf{of}\ \{[\,] \to ys;\ \ (a : x) \to a : \mathit{appendB}\ x\ ys\}, \\ \mathit{uncurryB} = \dots \end{array} \right\} \\
\Gamma_{\text{app}} &= \{\mathit{appendB} : [\mathsf{Int}] \to [\mathsf{Int}] \to [\mathsf{Int}], \mathit{uncurryB} : \dots\} \\
f_{1\text{app}} &= \mathit{uncurryB}\ \mathit{appendB} \\
\mathcal{E}_{\text{app}} &= \{(([1, 2, 3], [4, 5]), [6, 2], ([6, 2], []))\}.
\end{aligned}
$$

[4] We use metavariables $A, B, \dots$ for types in general and $\sigma, \tau, \dots$ for those that can be sources or views. In Matsuda and Wang [9] σ-types have a restriction that they do not contain $\mathbf{B}$ and $\to$, but their implementation does not distinguish A-types and σ-types (which in fact is safe). Thus, we do not strictly respect the restriction on σ-types in our technical development.

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 Sect. 2, we sometimes need to leave some parts unidirectional to achieve the 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 Example 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 $\mathbf{B}[\mathsf{Int} \to \mathsf{Int}]$.[5]

Next, we provide the typing environment generation relation $\Gamma \rightsquigarrow \Gamma'$, where Γ' is the typing environment corresponding to the bidirectional program.

Definition 1 (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 [9] uses a dual context system [22]. 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 definition of (our fragment of) HOBiT's typing relation is given in Fig. 2. The definition assumes constructors have simple types $\mathsf{C} : A_1 \to \cdots \to A_n \to A$, which may be instantiation results of their (rank-1) polymorphic types (such as list constructors (:) and [ ]).

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

[5] A function type $\to$ in the $\mathbf{B}$ type constructor is non-useful because HOBiT cannot handle it. In the put execution of HOBiT, it is necessary to check the consistency of variables' updated values that are used more than two times. However, the equality of functions is generally unclear. Thus, a $\mathbf{B}(* \to *)$ type term cannot be evaluated in the put direction. For the same reason, the nested $\mathbf{B}$ is non-useful.

$$\boxed{\Gamma;\ \Delta \vdash e : A}$$

$$\frac{\Gamma(x) = A}{\Gamma;\ \Delta \vdash x : A} \qquad \frac{\Delta(x) = \tau}{\Gamma;\ \Delta \vdash x : \mathbf{B}\tau} \qquad \frac{\Gamma, x : A;\ \Delta \vdash e : B}{\Gamma;\ \Delta \vdash \lambda x.e : A \to B}$$

$$\frac{\Gamma;\ \Delta \vdash e_1 : A \to B \quad \Gamma;\ \Delta \vdash e_2 : A}{\Gamma;\ \Delta \vdash e_1\ e_2 : B}$$

$$\frac{\{\Gamma;\ \Delta \vdash e : A_i\}_i \quad \mathsf{C} : A_1 \to \cdots \to A_n \to A}{\Gamma;\ \Delta \vdash \mathsf{C}\ \overline{e} : A}$$

$$\frac{\Gamma;\ \Delta \vdash e_0 : A \quad \{\Gamma_i \vdash p_i : A \quad \Gamma, \Gamma_i;\ \Delta \vdash e_i : B\}_i}{\Gamma;\ \Delta \vdash \mathbf{case}\ e_0\ \mathbf{of}\ \{p_i \to e_i\}_i : B}$$

$$\frac{\Gamma;\ \Delta \vdash e : \tau}{\Gamma;\ \Delta \vdash \,!e : \mathbf{B}\tau} \qquad \frac{\{\Gamma;\ \Delta \vdash e_i : \mathbf{B}\tau_i\}_i \quad \mathsf{C} : \tau_1 \to \cdots \to \tau_n \to \tau}{\Gamma;\ \Delta \vdash \underline{\mathsf{C}}\ \overline{e} : \mathbf{B}\tau}$$

$$\frac{\Gamma;\ \Delta \vdash e_0 : \mathbf{B}\tau \quad \left\{ \begin{array}{c} \Delta_i \vdash p_i : A \quad \Gamma;\ \Delta, \Delta_i \vdash e_i : B \\ \Gamma;\ \Delta \vdash e_i' : \sigma \to \mathsf{Bool} \quad \Gamma;\ \Delta \vdash e_i'' : \tau \to \sigma \to \tau \end{array} \right\}_i}{\Gamma;\ \Delta \vdash \underline{\mathbf{case}}\ e_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ e_i'\ \underline{\mathbf{by}}\ e_i''\}_i : \mathbf{B}\sigma}$$

$$\boxed{\Gamma \vdash p : A}$$

$$\frac{}{x : A \vdash x : A} \qquad \frac{\{\Gamma_i \vdash p_i : A_i\} \quad \mathsf{C} : A_1 \to \cdots \to A_n \to A}{\Gamma_1, \ldots, \Gamma_n \vdash \mathsf{C}\ \overline{p} : A}$$

$$\boxed{\Gamma \vdash P}$$

$$\frac{\{\Gamma;\ \emptyset \vdash e_i : \Gamma(f_i)\}_i}{\Gamma \vdash f_1 = e_1;\ \ldots;\ f_n = e_n}$$

Fig. 2 Typing rules: $\Delta \vdash p : \sigma$ is defined similarly to $\Gamma \vdash p : A$ but asserts that the resulting environment is actually an bidirectional one and every type that occurs in the derivation is a σ-type

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 $\Gamma';\ \Delta';\ A' \rhd \Gamma \vdash e : A \rightsquigarrow e'$; i.e., its outcome is only e'.

Figure 3 provides the sketch generation rules. For simplicity of presentation, we did not explicitly capture the type of the code to be generated in the specialized holes in the later sections as it can be recovered from the sketch and typing environment Γ. However, here we make it explicit by augmenting both the exit condition hole and the reconciliation hole with the typing environment Γ and the type of the code to be synthesized: $\square^{\mathrm{e}}(\Gamma, \sigma \to \mathsf{Bool}; e)$ and $\square^{\mathrm{r}}(\Gamma, \sigma_0 \to \sigma \to \sigma_0; p, e)$. Notice that $\square^{\mathrm{e}}$ and $\square^{\mathrm{r}}$ have the information of each branch's form (e and $p \to e$, respectively) to narrow the search space in the following synthesis process. If readers are interested in the treatment of let-polymorphism, please refer to Sec. 4.

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 2 (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 $\Gamma';\ \emptyset;\ \Gamma'(f_i) \rhd \Gamma \vdash e_i : \Gamma(f_i) \rightsquigarrow e_i'$. $\square$

$$\frac{\Gamma';\ \Delta';\ \tau \rhd \Gamma \vdash e : \tau \rightsquigarrow e'}{\Gamma';\ \Delta';\ \mathbf{B}\tau \rhd \Gamma \vdash e : \tau \rightsquigarrow\ !e'}\ (\text{G- !}) \qquad \frac{x : A \in \Gamma \quad x : A' \in \Gamma'}{\Gamma';\ \Delta';\ A' \rhd \Gamma \vdash x : A \rightsquigarrow x}\ (\text{G- UVAR})$$

$$\frac{x : \sigma \in \Gamma \quad x : \sigma \in \Delta'}{\Gamma';\ \Delta';\ \mathbf{B}\sigma \rhd \Gamma \vdash x : \sigma \rightsquigarrow x}\ (\text{G- BVAR})$$

$$\frac{(\Gamma', x : A_1');\ \Delta';\ A_2' \rhd (\Gamma, x : A_1) \vdash e : A_2 \rightsquigarrow e'}{\Gamma';\ \Delta';\ (A_1' \to A_2') \rhd \Gamma \vdash \lambda x.e : A_1 \to A_2 \rightsquigarrow \lambda x.e'}\ (\text{G- ABS})$$

$$\frac{\begin{array}{c}\Gamma';\ \Delta';\ (A_2' \to A') \rhd \Gamma \vdash e_1 : (A_2 \to A) \rightsquigarrow e_1' \quad A_2 \rightsquigarrow A_2' \\ \Gamma';\ \Delta';\ A_2' \rhd \Gamma \vdash e_2 : A_2 \rightsquigarrow e_2'\end{array}}{\Gamma';\ \Delta';\ A' \rhd \Gamma \vdash e_1\ e_2 : A \rightsquigarrow e_1'\ e_2'}\ (\text{G- APP})$$

$$\frac{\mathsf{C} : A_1 \to \cdots \to A_n \to A \quad \{\Gamma';\ \Delta';\ A_i \rhd \Gamma \vdash e_i : A_i \rightsquigarrow e_i'\}_i}{\Gamma';\ \Delta';\ A \rhd \Gamma \vdash \mathsf{C}\ \overline{e} : A \rightsquigarrow \mathsf{C}\ \overline{e'}}\ (\text{G- CON})$$

$$\frac{\mathsf{C} : \tau_1 \to \cdots \to \tau_n \to \tau \quad \{\Gamma';\ \Delta';\ \mathbf{B}\tau_i \rhd \Gamma \vdash e_i : \tau_i \rightsquigarrow e_i'\}_i}{\Gamma';\ \Delta';\ \mathbf{B}\tau \rhd \Gamma \vdash \mathsf{C}\ \overline{e} : \tau \rightsquigarrow \underline{\mathsf{C}}\ \overline{e'}}\ (\text{G- BCON})$$

$$\frac{\begin{array}{c}\Gamma';\ \Delta';\ A_0' \rhd \Gamma \vdash e_0 : A_0 \rightsquigarrow e_0' \quad A_0 \rightsquigarrow A_0' \\ \{\Gamma_i \vdash p_i : A_0 \quad \Gamma_i' \vdash p_i : A_0' \quad (\Gamma', \Gamma_i');\ \Delta';\ A' \rhd (\Gamma, \Gamma_i) \vdash e_i : A \rightsquigarrow e_i'\}_i\end{array}}{\Gamma';\ \Delta';\ A' \rhd \Gamma \vdash \textbf{case}\ e_0\ \textbf{of}\ \{p_i \to e_i\}_i : A \rightsquigarrow \textbf{case}\ e_0'\ \textbf{of}\ \{p_i \to e_i'\}_i}\ (\text{G- CASE})$$

$$\frac{\begin{array}{c}\Gamma';\ \Delta';\ \mathbf{B}\sigma_0 \rhd \Gamma \vdash e_0 : \sigma_0 \rightsquigarrow e_0' \\ \{\Gamma_i \vdash p_i : \sigma_0 \quad \Delta_i' \vdash p_i : \sigma_0 \quad \Gamma';\ (\Delta', \Delta_i');\ \mathbf{B}\sigma \rhd (\Gamma, \Gamma_i) \vdash e_i : \sigma \rightsquigarrow e_i'\end{array}}{\begin{array}{c}\Gamma';\ \Delta';\ \mathbf{B}\sigma \rhd \Gamma \vdash \textbf{case}\ e_0\ \textbf{of}\ \{p_i \to e_i\}_i : \sigma \\ \rightsquigarrow \underline{\textbf{case}}\ e_0'\ \underline{\textbf{of}} \left\{\begin{array}{l}\underline{p_i} \to e_i' \\ \underline{\textbf{with}}\ \square^{\mathrm{e}}(\Gamma', \sigma \to \mathsf{Bool};\ e_i') \\ \underline{\textbf{by}}\ \square^{\mathrm{r}}(\Gamma', \sigma_0 \to \sigma \to \sigma_0;\ p_i, e_i)\end{array}\right\}_i\end{array}}\ (\text{G- BCASE})$$

Fig. 3 Sketch generation rules for expressions

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 Sect. 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 by checking the form of the result. For example, for $\square^{\mathrm{e}}(a : \mathit{append}\ x\ \mathit{ys})$, 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):

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

For example, we have $\mathcal{P}(a : \mathit{append}\ x\ \mathit{ys}) = \mathcal{P}(a) : \mathcal{P}(\mathit{append}\ x\ \mathit{ys}) = z : \mathit{zs}$, 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 (Partial completion of exit condition holes) Let p_e be a pattern $\mathcal{P}(e)$. Then, the exit-condition-hole partial completion relation $\Box^{\mathrm{e}}(e) \rightsquigarrow e'$, which reads hole $\Box^{\mathrm{e}}(e)$ is filled by e', is defined by the rule

$$\Box^{\mathrm{e}}(e) \rightsquigarrow \lambda v.\, \mathbf{case}\ v\ \mathbf{of}\ \{p_e \to \Box;\ \_ \to \mathsf{False}\}$$

□

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

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

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 (Sect. 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. For example, consider the situation where $p = a : \mathit{as}$ and $\mathit{reverse}$ is provided as an auxiliary function. If we restrict the form of a hole as $\lambda s.\lambda \mathcal{P}(e).(\Box : \Box)$ according to the pattern, generating a term that behaves the same as $\lambda s.\lambda \mathcal{P}(e).\mathit{reverse}(s)$ needs an additional recursion. Thus, we generate reconciliation functions of the form $\lambda s.\lambda \mathcal{P}(e).\Box(p)$ instead. Recall that the shape-restricted hole $\Box(p)$ will be filled by expressions that may evaluate to values shaped p. This idea is formally written as below.

Definition 4 (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, $\Box^{\mathrm{r}}(p, e) \rightsquigarrow e'$, which reads hole $\Box^{\mathrm{r}}(p, e)$ is filled by e', is defined by the rule

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

□

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:

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

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 $K[\mathbf{case}\ e\ \mathbf{of}\ \{p_i \rightarrow e_i\}_i]$ can be hoisted as $\mathbf{case}\ e\ \mathbf{of}\ \{p_i \rightarrow K[e_i]\}_i$, 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 may be a component function that users provided.

The ability to generate **case**s in U is especially useful for synthesizing exit conditions for a branch where $\mathcal{P}(e)$ is not precise enough to distinguish branches. An example is *snoc* taken from our experiments (Sect. 5.1), where we will eventually obtain the following candidate after the sketch completion step I (the **by** parts are omitted).

$$
\begin{array}{l}
\mathit{snoc} :: \mathbf{B}[a] \rightarrow \mathbf{B}a \rightarrow \mathbf{B}[a] \\
\mathit{snoc}\ \mathit{xs}\ b = \underline{\mathbf{case}}\ \mathit{xs}\ \underline{\mathbf{of}} \\
\quad [\,] \quad\ \ \rightarrow b\ \underline{:}\ \underline{[\,]} \qquad\qquad\ \underline{\mathbf{with}}\ \lambda v.\mathbf{case}\ v\ \mathbf{of}\ \{[b] \rightarrow \square_1;\ \_ \rightarrow \mathsf{False}\} \\
\quad a : x \rightarrow a\ \underline{:}\ \mathit{snoc}\ x\ b\ \ \underline{\mathbf{with}}\ \lambda v.\mathbf{case}\ v\ \mathbf{of}\ \{a : r \rightarrow \square_2;\ \_ \rightarrow \mathsf{False}\}
\end{array}
$$

The most precise exit condition for the second branch is one that returns True for a list whose length is at least 2 and otherwise returns False. Such an exit condition is obtained by filling $\square_2$ with $\mathbf{case}\ r\ \mathbf{of}\ \{[\,] \rightarrow \mathsf{False};\ \_ : \_ \rightarrow \mathsf{True}\}$, which can be generated from U but not from V. This indeed is key to the success for *snoc* (Table 1) in our experiments, as the input/output examples given for *snoc* (Table 2) reject exit conditions obtained by filling $\square_2$ with True/False.

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.

It is worth noting that, though our term generation method is similar to Osera and Zdancewic [23], there are subtle differences. One of the differences is that we rely on the lazy nondeterministic generation [24], which delays generation until the investigation of the generation results. Another subtle difference is that we use p for filtering. This is for optimization. Recall that we synthesize a reconciliation function of the form $\lambda s.\lambda v.\,\textbf{case}\ v\ \textbf{of}\ \{\mathcal{P}(e) \to \Box(p)\}$ for a branch $p \to e$ by filling $\Box(p)$ with a term U^p, as the return value of the reconciliation function must match p. Since this constraint is orthogonal and common to all input/output examples, we fuse the filtering process to the generation process. This is especially useful for synthesizing a reconciliation function for a branch with a pattern involving complex constants such as `"section"`, as seen in XML queries examples examined in Sect. 5.2.1; note that the (even lazy) trial-and-error approach takes a vast amount of time to produce `"section"` as it can only be obtained after trying smaller strings than `"section"`. See Fig. 13 for a concrete synthesis result where this works effectively.

3.5.2 Filtering based on branch traces

As explained in the discussion on round-tripping in HOBiT (see the corresponding paragraph in Sect. 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 Sect. 3.1 only showed how filtering works for exit conditions, we provide another example illustrating filtering based on branch traces for both exit conditions and reconciliation functions. Then, we provide the formal definition of filtering based on branch traces. Finally, we conclude with a discussion on pruning away programs that would otherwise cause non-terminating *put* executions.

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

$$
\begin{array}{l}
f :: \mathsf{Either\ Int\ Int} \to \mathsf{Bool} \\
f\ x = \textbf{case}\ x\ \textbf{of}\ \{\mathsf{Left}\ x \to \mathsf{True};\ \ \mathsf{Right}\ x \to \mathsf{False}\}
\end{array}
$$

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

$$
\begin{array}{llll}
\multicolumn{4}{l}{f\ x = \underline{\textbf{case}}\ x\ \underline{\textbf{of}}} \\
\quad \mathsf{Left}\ x & \to \underline{\mathsf{True}} & \underline{\textbf{with}} & \lambda v.\textbf{case}\ v\ \textbf{of}\ \{\mathsf{True} \to \Box_1;\ \ \mathsf{False} \to \mathsf{False}\} \\
 & & \underline{\textbf{by}} & \lambda s.\lambda v.\textbf{case}\ v\ \textbf{of}\ \{\mathsf{True} \to \Box_2\} \\
\quad \mathsf{Right}\ x & \to \underline{\mathsf{False}} & \underline{\textbf{with}} & \lambda v.\textbf{case}\ v\ \textbf{of}\ \{\mathsf{False} \to \Box_3;\ \ \mathsf{True} \to \mathsf{False}\} \\
 & & \underline{\textbf{by}} & \lambda s.\lambda v.\textbf{case}\ v\ \textbf{of}\ \{\mathsf{False} \to \Box_4\}
\end{array}
$$

We first discuss filtering of exit conditions. For the first example, $\mathtt{:get}\ f\ (\mathsf{Right}\ 42)$ takes the second branch, meaning that we obtain the constraint $\Box_3[\mathsf{False}/v] \equiv \mathsf{True}$. For the second example, $\mathtt{:get}\ f\ (\mathsf{Left}\ 42)$ takes the first branch, generating the constraint $\Box_1[\mathsf{True}/v] \equiv \mathsf{True}$. A solution for these constraints is $\Box_1 = \mathsf{True}$ and $\Box_3 = \mathsf{True}$.

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

From these constraints, one possible solution is $\square_2 = \mathsf{Left}\ 42$ and $\square_3 = \mathsf{Right}\ 42$. While this solution obeys the given example, a better one would be $\square_2 = \textbf{case}\ s\ \textbf{of}\ \{\mathsf{Left}\ x \rightarrow s; \mathsf{Right}\ y \rightarrow \mathsf{Left}\ y\}$ and $\square_3 = \textbf{case}\ s\ \textbf{of}\ \{\mathsf{Left}\ x \rightarrow \mathsf{Right}\ x; \mathsf{Right}\ y \rightarrow s\}$; 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 $\texttt{:put}\,f\,(\mathsf{Left}\ 37, \mathsf{False}) = \mathsf{Right}\ 37$ were provided by the user.

As a note, for both the previous example and the running example *appendB* in Sect. 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 Sect. 5. In such a case, the choice of reconciliation functions may affect the generated constraints as they specify new sources.

Formalization of filtering based on branch traces Here, we provide details on the filtering based on branch traces. A trace is a tree that preserves information about which branches of $\underline{\textbf{case}}$ were chosen during the *get* or *put* evaluation. The syntax of such a trace is as follows:

$$tr ::= \epsilon \mid \mathrm{Br}(tr_0, j, tr_1) \mid [tr_1, \ldots, tr_n]$$

Here, $\mathrm{Br}(tr_0, j, tr_1)$ is a trace for $\underline{\textbf{case}}\ E_0\ \underline{\textbf{of}}\ \{p_i \rightarrow e_i\ \textbf{with}\ v_i'\ \textbf{by}\ v_i''\}_i$, where tr_0 is the trace for E_0, j stands for the j-th branch that was chosen, and tr_1 is the trace for E_j. The traces $tr_1, \ldots, tr_n$ in $[tr_1, \ldots, tr_n]$ correspond to the arguments of a constructor application $\underline{\mathsf{C}}\ e_1\ , \ldots,\ e_n$. We abbreviate $[tr_1, \ldots, tr_n]$ to $\overline{tr}$ when n is obvious.

Primer on HOBiT's formal semantics To understand the definition of filtering based on branch traces, one needs to know the formal semantics of HOBiT. The semantics of HOBiT is defined so that a term-in-context $\emptyset; \Delta \vdash e : \mathbf{B}\sigma$ defines a bidirectional transformation between Δ and σ. However, it is not clear how to interpret function abstractions and applications to achieve this goal. Thus, HOBiT adopts the staged semantics; it first evaluates λs away to obtain a first-order residual expression E from e, and then interprets E as a bidirectional transformation between Δ and σ. Specifically, we use three evaluation relations: $e \Downarrow v$ is for unidirectional evaluation to obtain residual expressions, and $\mu \vdash_{\mathrm{G}} E \Rightarrow u$ and $\mu \vdash_{\mathrm{P}} E \Leftarrow u \dashv \mu'$ for bidirectional evaluation of residual expressions. Intuitively, $\mu \vdash_{\mathrm{G}} E \Rightarrow u$ means that a residual expression E is evaluated in *get* direction to obtain the original view v from the the original source environment μ. Contrarily, $\mu \vdash_{\mathrm{PT}} E \Leftarrow_{tr} v \dashv \mu'$ means that a residual expression E is evaluated in *put* direction to obtain the updated source environment μ' from the the original source environment μ and the updated view v.

Values v, residual expressions E and first-order values u are defined as follows.

$$\begin{array}{lcl} v & ::= & \mathsf{True} \mid \mathsf{False} \mid [\,] \mid v_1 : v_2 \mid \lambda x.e \mid E \\ E & ::= & x \mid \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid E_1 \,\underline{:}\, E_2 \mid \textbf{case}\ E_0\ \textbf{of}\ \{\underline{p_i} \rightarrow e_i\ \textbf{with}\ v_i'\ \textbf{by}\ v_i''\}_i \mid \,!u \\ u & ::= & \mathsf{True} \mid \mathsf{False} \mid [\,] \mid u_1 : u_2 \end{array}$$

$$\frac{}{x \Downarrow x} \quad \frac{(f = e) \in P \quad e \Downarrow v}{f \Downarrow v} \quad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad e[v_2/x] \Downarrow v}{e_1\ e_2 \Downarrow v} \quad \frac{}{\lambda x.e \Downarrow \lambda x.e}$$

$$\frac{e_0 \Downarrow E_0 \quad \{e'_i \Downarrow v'_i \quad e''_i \Downarrow v''_i\}_i}{\textbf{case}\ e_0\ \textbf{of}\ \{\underline{p_i} \to e_i\ \textbf{with}\ e'_i\ \textbf{by}\ e''_i\}_i \Downarrow \textbf{case}\ E_0\ \textbf{of}\ \{\underline{p_i} \to e_i\ \textbf{with}\ v'_i\ \textbf{by}\ v''_i\}_i}$$

Fig. 4 Unidirectional evaluation rules (excerpt)

Intuitively, v, E, and u represent evaluation results of A-, $\mathbf{B}\tau$-, and τ-typed expressions, respectively. For more details on HOBiT, please see HOBiT's original paper [9].

SYNBIT's evaluation rules with traces The rules for the unidirectional evaluation relation are rather standard, as excerpted in Fig. 4. The bidirectional constructs (i.e., bidirectional constructors and bidirectional **case**) are frozen, i.e., treated as ordinary constructors in this evaluation.

For filtering based on trace branches we make use of the evaluation relations $\mu \vdash_{\text{GT}} E \Rightarrow_{tr} u$ and $\mu \vdash_{\text{PT}} E \Leftarrow_{tr} u \dashv \mu'$, defined by the rules in Fig. 5. Intuitively, similar to HOBiT's semantics, $\mu \vdash_{\text{GT}} E \Rightarrow_{tr} v$ means that a program E is reduced to the original view v under the the original source environment μ through a trace tr. Accordingly, $\mu \vdash_{\text{PT}} E \Leftarrow_{tr} v \dashv \mu'$ means that a program E is evaluated according to its *put* behavior such that the updated source environment μ' is obtained from the the original source environment μ and the updated view v through a trace tr. These traced evaluation relations only differ from the original *get* and *put* evaluation relations [9] in reference to traces. Thus, we omit the definitions of the original, untraced evaluation relations $\mu \vdash_{\text{G}} E \Rightarrow u$ and $\mu \vdash_{\text{P}} E \Leftarrow u \dashv \mu'$, even though they appear in Fig. 5.

In the evaluation rules, we write $[\![p]\!]$ for the semantics of pattern matching by p: i.e., $[\![p]\!]\ u$ is a partial function that returns a binding μ with $\text{dom}(\mu) = \text{fv}(p)$ and $u = p\mu$ if such μ exists (and fails otherwise). In HOBiT, $[\![p]\!]$ is injective and thus has a left inverse $[\![p]\!]^{-1}$. The rules for $\underline{\textbf{case}}$ (i.e., TG- BCASE, TP- BCASE1, and TP- BCASE2) implicitly assume that $\text{dom}(\mu_i)$ (i.e., $\text{fv}(p_i)$) and $\text{dom}(\mu)$ are disjoint; we assume appropriate α-renaming that is consistent in *get* and *put* to fulfill the condition. To separate environments, the evaluation rules use $\mu_1 \uplus_{X,Y} \mu_2$ which behaves similarly to the disjoint union $\mu_1 \uplus \mu_2$, while also ensuring $\text{dom}(\mu_1) \subseteq X$ and $\text{dom}(\mu_2) \subseteq Y$. The operator $\triangleleft$ is defined by:

$$(\mu' \triangleleft \mu)(x) = \begin{cases} \mu'(x) & \text{if } x \in \text{dom}(\mu') \\ \mu(x) & \text{if } x \in \text{dom}(\mu) \setminus \text{dom}(\mu') \end{cases}$$

The operator $\curlyvee$ is defined as: $\mu_1 \curlyvee \mu_2 = \mu_1 \cup \mu_2$ if $\mu_1(x) = \mu_2(x)$ for all $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, and otherwise $\mu_1 \curlyvee \mu_2$ is undefined.

Finally, we describe the filtering process using the rules shown in Fig. 5. Let $P' = \{f_i = e'_i\}_i$ be a synthesized program that will be checked by the filtering process, and let (s, v, s') be an input/output example of the backward behavior. We check whether `:put` $f_1\ s\ v = s'$ is established and its trace coincides with the trace of `:get` $f_1\ s' = v$ by the following procedure:

1. Let x be a fresh variable. Find E_1 that meets $e'_1\ x \Downarrow E_1$.
2. Obtain the trace tr that meets $\{x \mapsto s'\} \vdash_{\text{GT}} E_1 \Rightarrow_{tr} v$.
3. Calculate μ' that meets $\{x \mapsto s\} \vdash_{\text{PT}} v \Leftarrow_{tr} E_1 \dashv \mu'$. If there is no such μ', the synthesized program is inconsistent with the trace tr and should be filtered out.

$$\frac{}{\mu \vdash_{\mathrm{GT}} x \Rightarrow_{\epsilon} \mu(x)}\ (\textsc{Tg- Var}) \qquad \frac{}{\mu \vdash_{\mathrm{PT}} x \Leftarrow_{\epsilon} u \dashv \{x = u\}}\ (\textsc{Tp- Var})$$

$$\frac{\{\mu \vdash_{\mathrm{GT}} e_i \Rightarrow_{tr_i} u_i\}_i}{\mu \vdash_{\mathrm{GT}} \underline{\mathsf{C}}\ \overline{e} \Rightarrow_{\overline{tr}} \mathsf{C}\ \overline{u}}\ (\textsc{Tg- BCon}) \qquad \frac{\{\mu \vdash_{\mathrm{PT}} e_i \Leftarrow_{tr_i} u_i \dashv \mu'_i\}_i}{\mu \vdash_{\mathrm{PT}} \underline{\mathsf{C}}\ \overline{e} \Leftarrow_{\overline{tr}} \mathsf{C}\ \overline{u} \dashv \bigcurlyvee \mu'_i}\ (\textsc{Tp- BCon})$$

$$\frac{}{\mu \vdash_{\mathrm{GT}} !u \Rightarrow_{\epsilon} u}\ (\textsc{Tg- !}) \qquad \frac{}{\mu \vdash_{\mathrm{PT}} !u \Leftarrow_{\epsilon} u \dashv \emptyset}\ (\textsc{Tp- !})$$

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{GT}} E_0 \Rightarrow_{tr_0} u_0 \quad [\![p_i]\!]\ u_0 = \mu_i \quad e_i \Downarrow E_i \\ \mu \uplus \mu_i \vdash_{\mathrm{GT}} E_i \Rightarrow_{tr_i} u \quad v'_i\ u \Downarrow \mathsf{True}\end{array}}{\mu \vdash_{\mathrm{GT}} \underline{\mathbf{case}}\ E_0\ \mathbf{of}\ \{\underline{p_i} \to e_i\ \underline{\mathbf{with}}\ v'_i\ \underline{\mathbf{by}}\ v''_i\} \Rightarrow_{\mathrm{Br}(tr_0, i, tr_i)} u}\ (\textsc{Tg- BCase})$$

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{G}} E_0 \Rightarrow u_0 \quad [\![p_i]\!]\ u_0 = \mu_i \quad v'_i\ u \Downarrow \mathsf{True} \\ e_i \Downarrow E_i \quad \mu \uplus \mu_i \vdash_{\mathrm{PT}} E_i \Leftarrow_{tr_i} u \dashv \mu' \uplus_{\mathrm{dom}(\mu),\mathrm{dom}(\mu_i)} \mu'_i \\ [\![p_i]\!]^{-1}\ (\mu'_i \lhd \mu_i) = u'_0 \quad \mu \vdash_{\mathrm{PT}} E_0 \Leftarrow_{tr_0} u'_0 \dashv \mu'_0\end{array}}{\mu \vdash_{\mathrm{PT}} \underline{\mathbf{case}}\ E_0\ \mathbf{of}\ \{\underline{p_i} \to e_i\ \underline{\mathbf{with}}\ v'_i\ \underline{\mathbf{by}}\ v''_i\} \Leftarrow_{\mathrm{Br}(tr_0, i, tr_i)} u \dashv \mu'_0 \curlyvee \mu'}\ (\textsc{Tp- BCase1})$$

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{G}} E_0 \Rightarrow u_0 \quad [\![p_i]\!]\ u_0 = \mu_i \quad v'_i\ u \Downarrow \mathsf{False} \quad v'_j\ u \Downarrow \mathsf{True} \\ \{v'_k\ u \Downarrow \mathsf{False}\}_{k<j} \quad v''_j\ u_0\ u \Downarrow u_0^{\mathrm{rec}} \quad [\![p_j]\!]\ u_0^{\mathrm{rec}} = \mu_j \\ e_j \Downarrow E_j \quad \mu \uplus \mu_j \vdash_{\mathrm{PT}} E_j \Leftarrow_{tr_i} u \dashv \mu' \uplus_{\mathrm{dom}(\mu),\mathrm{dom}(\mu_j)} \mu'_j \\ [\![p_j]\!]^{-1}\ (\mu'_j \lhd \mu_j) = u'_0 \quad \mu \vdash_{\mathrm{PT}} E_0 \Leftarrow_{tr_0} u'_0 \dashv \mu'_0\end{array}}{\mu \vdash_{\mathrm{PT}} \underline{\mathbf{case}}\ E_0\ \mathbf{of}\ \{\underline{p_i} \to e_i\ \underline{\mathbf{with}}\ v'_i\ \underline{\mathbf{by}}\ v''_i\} \Leftarrow_{\mathrm{Br}(tr_0, j, tr_i)} u \dashv \mu'_0 \curlyvee \mu'}\ (\textsc{Tp- BCase2})$$

Fig. 5 *get* and *put* evaluation rules with traces

4. Check whether $(\mu' \lhd \{x \mapsto s\})(x)$ equals to s', which means $\texttt{:put}\ f_1\ s\ v = s'$. If established, the candidate program is consistent with the given input/output example.

For illustration, we use *appendB*, and assume the following example was given as the backward behavior:

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

The trace of

$$\texttt{:get}\ (\mathit{uncurryB}\ \mathit{appendB})\ ([6,2],[\,]) = [6,2]$$

and the trace of the above `:put` must coincide. The trace of this `:get` (after evaluating *uncurryB*) is obtained as follows:

$$\{xs = [6,2], ys = [\,]\} \vdash_{\mathrm{GT}} \mathit{appendBbody} \Rightarrow_{\mathrm{Br}(\epsilon,1,[\epsilon,\mathrm{Br}(\epsilon,1,[\epsilon,\mathrm{Br}(\epsilon,0,\epsilon)])])} [6,2]$$

where

$$\begin{array}{rlll} \mathit{appendBbody} = \underline{\mathbf{case}}\ xs\ \underline{\mathbf{of}}\ [\,] & \to ys & & \\ & \quad \underline{\mathbf{with}}\ \mathit{const}\ \mathsf{True} & \underline{\mathbf{by}}\ \lambda\_.\lambda\_.\,[\,] \\ a : x & \to a\ \underline{:}\ \mathit{appendB}\ x\ ys & \\ & \quad \underline{\mathbf{with}}\ \mathit{not} \circ \mathit{null} & \underline{\mathbf{by}}\ \lambda s.\lambda\_.\,s \end{array}$$

Here, the resulting overall trace is

$$tr = \mathrm{Br}^1(\epsilon^2, 1, [\epsilon^3, \mathrm{Br}^4(\epsilon^5, 1, [\epsilon^6, \mathrm{Br}^7(\epsilon^8, 0, \epsilon^9)])])$$

(for easy identification, we numbered the individual traces). Next, we explain how this is obtained. The first step of the evaluation of *get* is the case branch. Since we have $xs = [6, 2]$, the second branch (cons case) is chosen. Thus, by TG- BCASE (reproduced below),

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{GT}} E_0 \Rightarrow_{tr_0} u_0 \quad [\![p_i]\!]\, u_0 = \mu_i \quad e_i \Downarrow E_i \\ \mu \uplus \mu_i \vdash_{\mathrm{GT}} E_i \Rightarrow_{tr_i} u \quad v_i'\ u \Downarrow \mathsf{True}\end{array}}{\mu \vdash_{\mathrm{GT}} \mathbf{case}\ E_0\ \mathbf{of}\ \{\underline{p_i} \to e_i\ \mathbf{with}\ v_i'\ \mathbf{by}\ v_i''\} \Rightarrow_{\mathrm{Br}(tr_0,i,tr_i)} u}\ (\textsc{Tg- BCase})$$

the whole trace tr has the form of $\mathrm{Br}^1(tr_2, 1, tr_3)$, where tr_2 is the trace of the evaluation of xs, and tr_3 is the trace of the evaluation of $a \underline{:} appendB\ x\ ys$ under $\{a = 6, x = [2], ys = [\,], \dots\}$. By TG- VAR (reproduced below), we have $tr_2 = \epsilon^2$,

$$\frac{}{\mu \vdash_{\mathrm{GT}} x \Rightarrow_{\epsilon} \mu(x)}\ (\textsc{Tg- Var})$$

and by TG- BCON (reproduced below), we have $tr_3 = [tr_4, tr_5]$ for some trace tr_4 and tr_5.

$$\frac{\{\mu \vdash_{\mathrm{GT}} e_i \Rightarrow_{tr_i} u_i\}_i}{\mu \vdash_{\mathrm{GT}} \underline{\mathsf{C}}\ \overline{e} \Rightarrow_{\overline{tr}} \mathsf{C}\ \overline{u}}\ (\textsc{Tg- BCon})$$

Since tr_4 is the trace of the evaluation of a, we have $tr_4 = \epsilon^3$ by TG- VAR. Then, we focus on tr_5, which is the trace of the evaluation of $appendB\ x\ ys$ under $\{x = [2], ys = [\,], \dots\}$. Thus, we have[6]

$$\{xs = [2], ys = [\,], \dots\} \vdash_{\mathrm{GT}} appendBbody \Rightarrow_{tr_5} v_5$$

for some value v_5. The second branch (cons case) is again chosen, and we obtain $tr_5 = \mathrm{Br}^4(\epsilon^5, 1, [\epsilon^6, tr_6])$ for some trace tr_6. Since tr_6 is the trace of the evaluation of $appendB\ x\ ys$ under $\{x = [\,], ys = [\,], \dots\}$, we have

$$\{xs = [\,], ys = [\,], \dots\} \vdash_{\mathrm{GT}} appendBbody \Rightarrow_{tr_6} v_6$$

for some value v_6. Since we have $xs = [\,]$, the first branch (nil case) is chosen. Thus, by TG- BCASE, we have $tr_6 = \mathrm{Br}^7(tr_7, 0, tr_8)$ for some tr_7 and tr_8. Since tr_7 is the trace of the evaluation of xs, we have $tr_7 = \epsilon^8$ by TG- VAR. By TG- VAR, we have $tr_8 = \epsilon^9$. As a result, the whole trace of the *get* direction is computed as:

$$tr = \mathrm{Br}^1(\epsilon^2, 1, [\epsilon^3, \mathrm{Br}^4(\epsilon^5, 1, [\epsilon^6, \mathrm{Br}^7(\epsilon^8, 0, \epsilon^9)])])$$

Then, we check if `:put` $(uncurryB\ appendB)$ $([1, 2, 3], [4, 5])$ $[6, 2]$ goes through the same trace tr as *get*. Essentially, this means that the following relation must hold:

$$\begin{array}{l}\{xs = [1, 2, 3], ys = [4, 5]\} \vdash_{\mathrm{PT}} appendBbody \\ \Leftarrow_{\mathrm{Br}(\epsilon,1,[\epsilon,\mathrm{Br}(\epsilon,1,[\epsilon,\mathrm{Br}(\epsilon,0,\epsilon)])])}\ [6, 2] \dashv \mu'\end{array} \qquad (\dagger)$$

with $\mu' = \{xs = [6, 2], ys = [\,]\}$. In this evaluation, we also make use of the trace to determine the rule to be applied. For example, in the backward evaluation of *appendBbody* above, we know from the trace that the second branch (cons case) is chosen. Also, we know from the original environment, which maps xs to $[1, 2, 3]$, that branch switching does not happen. Thus, the last rule used in the evaluation must be TP- BCASE1 (reproduced below).

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{G}} E_0 \Rightarrow u_0 \quad [\![p_i]\!]\, u_0 = \mu_i \quad v_i'\ u \Downarrow \mathsf{True} \\ e_i \Downarrow E_i \quad \mu \uplus \mu_i \vdash_{\mathrm{PT}} E_i \Leftarrow_{tr_i} u \dashv \mu' \uplus_{\mathrm{dom}(\mu),\mathrm{dom}(\mu_i)} \mu_i' \\ [\![p_i]\!]^{-1}\ (\mu_i' \triangleleft \mu_i) = u_0' \quad \mu \vdash_{\mathrm{PT}} E_0 \Leftarrow_{tr_0} u_0' \dashv \mu_0'\end{array}}{\mu \vdash_{\mathrm{PT}} \mathbf{case}\ E_0\ \mathbf{of}\ \{\underline{p_i} \to e_i\ \mathbf{with}\ v_i'\ \mathbf{by}\ v_i''\} \Leftarrow_{\mathrm{Br}(tr_0,i,tr_i)} u \dashv \mu_0' \curlyvee \mu'}\ (\textsc{Tp- BCase1})$$

[6] If we follow the evaluation rules precisely, we will evaluate $appendBbody[x/xs, ys/ys]$ with α-renaming of a and x bound in *appendBbody*, which would bring further complication and thus is ignored here.

Then, we need to check the following conditions that appear as the premises of the rule (we shall omit the *get* evaluation of scrutinee and the obvious pattern matching as they are not relevant here).

$$(not \circ null)\ [6,2] \Downarrow \mathsf{True}$$

$$\begin{aligned}&\{a = 1, x = [2,3], ys = [4,5], \dots\} \vdash_{\mathrm{PT}} a\ \underline{:}\ appendB\ x\ ys \\ &\Leftarrow_{tr_3} [6,2] \dashv \mu_1' \uplus_{\{ys,\dots\},\{a,x\}} \mu_2'\end{aligned} \tag{$*$}$$

$$\begin{aligned}&\{xs = [1,2,3], \dots\} \vdash_{\mathrm{PT}} xs \\ &\Leftarrow_{tr_2} [\![a : x]\!]^{-1}(\mu_2' \triangleleft \{a = 1, x = [2,3]\}) \dashv \mu_3'\end{aligned}$$

$$\mu' = \mu_1' \curlyvee \mu_3'$$

Here, $tr_2, tr_3, \dots$ are the traces that appeared in the *get* evaluation; hence $tr_2 = \epsilon$ and $tr_3 = [tr_4, tr_5] = [\epsilon, \mathrm{Br}(\epsilon, 1, [\epsilon, \mathrm{Br}(\epsilon, 0, \epsilon)])]$ for example. The first condition clearly holds, and the third condition can be solved by TP- VAR (reproduced below) with $\mu_3' = \{xs = [\![a : x]\!]^{-1}(\mu_2' \triangleleft \{a = 1, x = [2,3]\})\}$.

$$\frac{}{\mu \vdash_{\mathrm{PT}} x \Leftarrow_{\epsilon} u \dashv \{x = u\}}\ (\text{TP- VAR})$$

So, let's focus on the second condition ($*$). By TP- BCON (reproduced below),

$$\frac{\{\mu \vdash_{\mathrm{PT}} e_i \Leftarrow_{tr_i} u_i \dashv \mu_i'\}_i}{\mu \vdash_{\mathrm{PT}} \underline{\mathsf{C}}\ \overline{e} \Leftarrow_{\overline{tr}} \mathsf{C}\ \overline{u} \dashv \bigcurlyvee \mu_i'}\ (\text{TP- BCON})$$

this condition is reduced to

$$\{a = 1, \dots\} \vdash_{\mathrm{PT}} a \Leftarrow_{tr_4} 6 \dashv \mu_4'$$

and checking whether tr_5 is the trace of the backward evaluation of *append x ys* under $\{x = [2,3], ys = [4,5], \dots\}$ for the updated view [2] to yield μ_5' such that $\mu_1' \uplus_{\{xs,ys\},\{a,x\}} \mu_2' = \mu_4' \curlyvee \mu_5'$. By TP- VAR, it is easy to see that the former holds with $\mu_4' = \{a = 6\}$. The latter holds if (and only if)[7]

$$\{xs = [2,3], ys = [4,5], \dots\} \vdash_{\mathrm{PT}} appendBbody \Leftarrow_{tr_5} [2] \dashv \mu_6'$$

holds with $tr_5 = \mathrm{Br}(\epsilon, 1, [\epsilon, tr_6]) = \mathrm{Br}(\epsilon, 1, [\epsilon, \mathrm{Br}(\epsilon, 0, \epsilon)])$ and $\mu_5' = \{x = \mu_6'(xs), ys = \mu_6'(ys)\}$. Again, in the backward evaluation of *appendBbody*, we know from the trace tr_5 that the second branch (cons case) must be chosen, and from the original environment $\{xs = [2,3], \dots\}$ that the branch is the original one. Thus, the last rule used for the evaluation must be TP- BCASE1. Thus, to make the above traced backward evaluation hold, it suffices (and needs) that the following conditions hold (again, we shall ignore irrelevant premises of TP- BCASE1).

$$(not \circ null)\ [2] \Downarrow \mathsf{True}$$

$$\begin{aligned}&\{a = 2, x = [3], ys = [4,5], \dots\} \vdash_{\mathrm{PT}} a\ \underline{:}\ appendB\ x\ ys \\ &\Leftarrow_{[\epsilon, tr_6]} [2] \dashv \mu_7' \uplus_{\{ys,\dots\},\{a,x\}} \mu_8'\end{aligned} \tag{$**$}$$

$$\{xs = [2,3], \dots\} \vdash_{\mathrm{PT}} xs \Leftarrow_{\epsilon} [\![a : x]\!]^{-1}(\mu_8' \triangleleft \{a = 2, x = [3]\}) \dashv \mu_9'$$

$$\mu_6' = \mu_7' \curlyvee \mu_9'$$

[7] Similar to Footnote 6.

The first condition clearly holds, and the third condition can be solved by TP- VAR with $\mu'_9 = \{xs = [\![a : x]\!]^{-1}(\mu'_8 \lhd \{a = 2, x = [3]\})$. The second condition (**) is further reduced to conditions:

$$\begin{gathered}
\{a = 2, \dots\} \vdash_{\mathrm{PT}} a \Leftarrow_{\epsilon} 2 \dashv \mu'_{10} \\
\{xs = [3], ys = [4,5], \dots\} \vdash_{\mathrm{PT}} \mathit{appendBbody} \Leftarrow_{tr_6} [\,] \dashv \mu'_{11} \\
\mu'_7 \uplus_{\{ys,\dots\},\{a,x\}} \mu'_8 = \mu'_{10} \curlyvee \{x = \mu'_{11}(xs), ys = \mu'_{11}(ys)\}.
\end{gathered}$$

Clearly, by TP- VAR, the first condition holds with $\mu'_{10} = \{a = 2\}$. Thus, let's focus on the second condition. Unlike the previous cases, the trace $tr_6 = \mathrm{Br}(\epsilon, 0, \epsilon)$ says that the evaluation must use the first branch (nil case), which is different from the original branch (taken under the environment $\{xs = [3], \dots\}$). Thus, the last rule used in the evaluation $\{xs = [3], ys = [4,5], \dots\} \vdash_{\mathrm{PT}} \mathit{appendBbody} \Leftarrow_{tr_6} [\,] \dashv \mu'_{11}$ must be TP- BCASE2 (reproduced below), meaning that branch switching happened.

$$\frac{\begin{gathered}
\mu \vdash_{\mathrm{G}} E_0 \Rightarrow u_0 \quad [\![p_i]\!]\, u_0 = \mu_i \quad v'_i\ u \Downarrow \mathsf{False} \quad v'_j\ u \Downarrow \mathsf{True} \\
\{v'_k\ u \Downarrow \mathsf{False}\}_{k<j} \quad v''_j\ u_0\ u \Downarrow u_0^{\mathrm{rec}} \quad [\![p_j]\!]\, u_0^{\mathrm{rec}} = \mu_j \\
e_j \Downarrow E_j \quad \mu \uplus \mu_j \vdash_{\mathrm{PT}} E_j \Leftarrow_{tr_i} u \dashv \mu' \uplus_{\mathrm{dom}(\mu),\mathrm{dom}(\mu_j)} \mu'_j \\
[\![p_j]\!]^{-1}\, (\mu'_j \lhd \mu_j) = u'_0 \quad \mu \vdash_{\mathrm{PT}} E_0 \Leftarrow_{tr_0} u'_0 \dashv \mu'_0
\end{gathered}}{\mu \vdash_{\mathrm{PT}} \underline{\mathbf{case}}\ E_0\ \mathbf{of}\ \{\underline{p_i} \to e_i\ \underline{\mathbf{with}}\ v'_i\ \underline{\mathbf{by}}\ v''_i\} \Leftarrow_{\mathrm{Br}(tr_0,j,tr_i)} u \dashv \mu'_0 \curlyvee \mu'} \ (\text{TP- BCASE2})$$

Since we have

$$(\lambda\_.\lambda\_.\,[\,])\ [3]\ [\,] \Downarrow [\,]$$

we then check other premises of the rule instance (again, we shall ignore the irrelevant ones):

$$\begin{gathered}
(\mathit{not} \circ \mathit{null})\ [\,] \Downarrow \mathsf{False} \\
(\mathit{const}\ \mathsf{True})\ [\,] \Downarrow \mathsf{True} \\
\{ys = [4,5], \dots\} \vdash_{\mathrm{PT}} ys \Leftarrow_{\epsilon} [\,] \dashv \mu'_{12} \uplus_{\{ys,\dots\},\emptyset} \emptyset \\
\{xs = [3], \dots\} \vdash_{\mathrm{PT}} xs \Leftarrow_{\epsilon} [\![[\,]]\!]^{-1}(\emptyset \lhd \emptyset) \dashv \mu'_{13} \\
\mu'_{11} = \mu'_{12} \curlyvee \mu'_{13}.
\end{gathered}$$

The first two clearly hold, and by TP- VAR, the third and forth ones hold with $\mu'_{12} = \{ys = [\,]\}$ and $\mu'_{13} = \{xs = [\,]\}$. Thus, we have $\mu'_{11} = \{xs = [\,], ys = [\,]\}$ to satisfy the fifth condition. Then, we can solve the constraints on environments obtained so far as: $\mu'_8 = \{a = 2, x = [\,]\}$, $\mu'_7 = \{ys = [\,]\}$, $\mu'_9 = \{xs = [2]\}$, $\mu'_6 = \{xs = [2], ys = [\,]\}$, $\mu'_5 = \{x = [2], ys = [\,]\}$, $\mu'_1 = \{ys = [\,]\}$, $\mu'_2 = \{a = 6, x = [2]\}$, and $\mu'_3 = \{xs = [6,2]\}$. Now, we are ready to check $\mu' = \{xs = [6,2], ys = [\,]\} = \mu'_1 \curlyvee \mu'_3$ to conclude (†).

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 [25], 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 [24], 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, $\mathsf{True}\,\&\&\,e$ 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 $x\ V_1\ \dots\ V_n$ 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; recall that the last step involves the *get* execution to obtain a trace to filter out the *put* behavior. 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 (Sect. 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 Sect. 5 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 Sect. 5.1.

4 Discussion on let-polymorphism

SYNBIT supports **let**-polymorphism as illustrated by some *get* programs used in the experiments (Sect. 5.1), such as $\mathit{append} :: [a] \to [a] \to [a]$ and $\mathit{reverse} :: [a] \to [a]$ that have polymorphic types. However, for simplicity, we assumed simple types in our formal development (e.g., Fig. 2 and 3).

The handling of bidirectional programs with (rank 1) polymorphic types is mostly straightforward. For example, for the sketch generation in Fig. 3, we can leverage the fact that in the standard typing system for **let** polymorphism [26], expressions can only have monomorphic types. As a consequence, we can say A and A' in $\Gamma'; \Delta'; A' \rhd \Gamma \vdash e : A \rightsquigarrow e'$ are always monomorphic. Also, these types A and A' are fully determined in advance of the generation, because A comes from the typing derivation of the original *get* program and A' is obtained from A by inserting **B**. The types A and A' may contain type variables but they are rigid (i.e., not subject to unification). Note that we can access to the typing derivation of the original *get* program as well as the code itself.

Though it is still straightforward, a slightly more careful discussion is needed for the type-directed generation of terms to fill shape-restricted holes (Sect. 3.5.1), where types being used for type-directed generation may not be immediately clear. In an extreme case, for example, if the component functions contain $\mathit{const} :: a \to b \to a$, we have no immediate information of the type to generate its second argument as it cannot be determined by instantiation of a. This suggests that we may need to consider explicit handing of type variables and unification in the term generation, which complicates the system. Fortunately though, our back-end system for (lazy) nondeterministic generation is powerful enough to address the issue without introducing any additional complication; in the system, a logic (i.e., unifiable) variable (e.g., (non-rigid) type variable) can be expressed by nondeterministic generation of all the things (e.g., monomorphic types) to substitute for it, and by the *share* operator, which is for sharing (non-deterministic) computational results [24].

5 Experiments

We implemented the proposed idea as a proof-of-concept system, SYNBIT, in Haskell.[8] SYNBIT is given as an extension to the original HOBiT implementation [9].

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

- Microbenchmarks, classified in terms of information loss (Sect. 5.1).
- More realistic problems including XML transformations and string parsing (Sect. 5.2).
- Comparisons with the other state-of-the-art synthesis methods (Sect. 5.3).

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

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

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,[9] a popular library in Haskell for benchmarking, which estimates the true execution time by the least-squares method. Any case running longer than 10 min was reported as a timeout.

5.1 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 lenses 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.[10] 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 function marked by $(*)$ includes a small adaptation to its standard definition which will be explained below. The definitions of all the functions listed and the full synthesis results can be found in the artifact[11] or the repository[12] 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

[9] https://hackage.haskell.org/package/criterion

[10] unless we know that the second list is fixed as in *appendBc*

[11] https://doi.org/10.5281/zenodo.5494504

[12] https://github.com/masaomi-yamaguchi/synbit

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	–

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 [27, 28] 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, 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.

5.2 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.

5.2.1 XML transformations

We examined six queries from XML Query Use Cases[13] ("TREE" Use Case). Table 3 provides brief explanations for these queries and Fig. 6 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.

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 "LOC_{in}" and "LOC_{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 the main body of this paper. See Appendix 8.2 for the concrete input and output of SYNBIT for Q1, which serves as a representative of the six to illustrate their complexity.)

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.

5.2.2 Lexer and parser

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

data Token = TNum Nat | LPar | RPar | Plus

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

Table 2 Input/output examples: for readability, we shall write n for $\mathsf{S}^n\ \mathsf{Z}$ (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	2	6	3
	5	10	4	2
replicate	('a', 2)	"aa"	"bbb"	('b', 3)
	(True, 3)	[True, True, True]	[False,False]	(False, 2)
mapNot	[True, False]	[False, True]	[True, True, False]	[False, False, True]
mapReplicate	[('b', 2)]	["bb"]	["aaa", "b", "cc"]	[('a', 3), ('b', 1), ('c', 2)]
	[('a', 3), ('b', 1), ('c', 2)]	["aaa", "b", "cc"]	["bb"]	[('b', 2)]
snoc	([1, 2, 3], 4)	[1, 2, 3, 4]	[1, 2, 3]	([1, 2], 3)
	([1, 2, 3], 4)	[1, 2, 3, 4]	[1, 2, 3, 4, 5, 6]	([1, 2, 3, 4, 5], 6)
length	[1, 2]	2	4	[1, 2, 0, 0]
/lengthTail	[2, 0]	2	1	[2]
reverse	[True, True]	[True, True]	[False, True, True]	[True, True, False]
	[1, 2, 3, 4]	[4, 3, 2, 1]	[6, 5]	[5, 6]
mapFst	[(1, 'a'), (2, 'b'), (3, 'c')]	[1, 2, 3]	[1, 3]	[(1, 'a'), (3, 'b')]
	[(2, 'b'), (3, 'c')]	[2, 3]	[0, 1, 2, 3]	[(0, 'b'), (1, 'c'), (2, 'a'), (3, 'a')]
add	(2, 3)	5	7	(2, 5)
	(2, 3)	5	1	(1, 0)
append	([1, 2, 3, 4], [5])	[1, 2, 3, 4, 5]	[6, 2]	([6, 2], [])
	([1, 2, 3, 4], [5])	[1, 2, 3, 4, 5]	[1, 2, 3, 4, 5, 6]	([1, 2, 3, 4], [5, 6])
appendBc	"apple"	"apple;;"	"pineapple;;"	"pineapple"
	"apple"	"apple;;"	"plum;;"	"plum"
professor	$[st_1, st_2, pr_1, st_3, pr_2]$	$[pr_1, pr_2]$	$[pr'_1, pr'_2, pr'_3]$	$[st_1, st_2, pr'_1, st_3, pr'_2, pr'_3]$
	$[st_1, st_2, pr_1, st_3, pr_2]$	$[pr_1, pr_2]$	$[pr'_1]$	$[st_1, st_2, pr'_1, st_3]$
lines	"aa\nbb\ncc"	["aa", "bb", "cc"]	["aa", "bb"]	"aa\nbb"
	"aa"	["aa"]	["aa", "bb"]	"aa\nbb"
	"aa\n"	["aa"]	["aa", "bb"]	"aa\nbb\n"
lookup	([(1, 10), (2, 200), (3, 33)], 2)	200	10	([(1, 10), (2, 200), (3, 33)], 1)
	([(1, 10), (2, 200), (3, 33)], 2)	200	33	([(1, 10), (2, 200), (3, 33)], 3)

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.”

```
<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. 6 An XML document used as an original source for Queries Q1 to Q6

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.[14] Sometimes, though, such lost information is attached to the abstract syntax trees, making the parsing process injective [29–31].

Table 5 summarizes the experimental results. In Fig. 7, 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. Please refer to Appendix 8.3 to see the concrete input and output of SYNBIT.

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

Table 4 The results of experiments for XML examples

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

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	0.094	69	265
			Remove/insert tokens			
Parser	10	45	Replacement of whole AST and back	0.55	29	121

```
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. 7 Synthesized Bidirectional Parser

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

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

5.3 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 [32–34], the state of the art lens synthesizer, as its inputs and outputs are too different from ours (see Sect. 6 for a non-experimental comparison).

Instead, we pick SMYTH [13], 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" [13], 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[15] or the repository[16]). 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.

6 Related work

Optician Optician [32–34] is the state-of-the-art framework for synthesizing lenses [4, 6, 35, 36]. Both their framework and ours implicitly guarantee the round-tripping properties

[15] https://doi.org/10.5281/zenodo.5494504

[16] https://github.com/masaomi-yamaguchi/synbit

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 8.1 for an illustrative example on the difficulty).

Despite the very different approaches, it is interesting to observe a common design principle shared by both: leveraging 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 [34] is able to prioritize the 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 Both the proposed approach and optician do not support matching lenses [37], which limits synthesized bidirectional transformations. One might think that the I/O example for *mapFst* in Table 2 is a bit unnatural and that it should behave as follows instead.

Original source	(Original view)	Updated view	Updated source
[(1, `'a'`), (2, `'b'`), (3, `'c'`)]	[1, 2, 3]	[1, 3]	[(1, `'a'`), (3, `'c'`)]

To achieve this goal, we need to know the correspondence between 3 in the view and (3, `'c'`) in the source, or *matching* [37] of elements. Finding appropriate matching is known to be the *alignment problem* in this context, which can typically be solved by using keys that identity elements in a container. For example, in the above I/O example, one implicitly assumes that the first component of a pair in the source list plays as a key, and 3 in the view and (3, `'c'`) should match as they have the same key 3. Currently, neither HOBiT used in SYNBIT nor the lenses [4, 6, 35, 36] used in Optician do not handle key constraints.

In a vision paper, Voigtländer [38] 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 [39] 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 [16, 17, 40–44]. A more interesting connection is the concept of *partial* inversion [45], 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 (Sect. 3.3), with the lazy nondeterministic generation, can be viewed as a type-based binding-time analysis [46]. The idea of partial inversion is further extended so that the return values of inverses are treated as "static inputs" as well [47], and the *pin operator* [48] 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. [5], based on the constant-complement view updating [1], 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 [49] (only variables can be arguments of functions) so that the injectivity analysis becomes exact. Voigtländer [50] makes use of parametricity [51, 52] to interpret polymorphic functions as bidirectional transformations. The technique is restricted to polymorphic functions. 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 [10, 53, 54]. 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 [15]. As opposed to our technique, Solar-Lezama's technique [15] 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) [55], 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 [20, 56–60]. Often, such approaches expect formal specifications describing the behavior of the target program, which can be difficult to write or expensive to check against using automated verification techniques. Conversely, our specification consists of the unidirectional program and input/output examples 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 Katayama [61], Feser et al. [62], Osera and Zdancewic [23], Lubin et al. [13], 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 [20, 21], 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. [25] 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 [63] generate only programs in a specific normal form, where term rewriting is used to transform a program into its normal form. In [64], 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 Sect. 3.6 worked well. However, we do plan on exploring some of the equivalence reduction techniques discussed here as future work.

7 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 Sect. 3.7. Another future direction is to extend the target language (HOBiT) based on our experience in order to synthesize more bidirectional transformations.

Acknowledgements 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. We also thank the members of the Bristol Programming Languages research group for creating the wonderful environment for this work. This work was partially supported by Japan Society for the Promotion of Science KAKENHI JP15H02681, JP19K11892, JP20H04161, and JP22H03562, Japan Society for the Promotion of Science Bilateral Program JPJSBP120199913, the Kayamori Foundation of Informational Science Advancement, Royal

Society University Research Fellowship UF160079, Royal Society grant IES\R3\170104, and Engineering and Physical Sciences Research Council grant EP/T008911/1.

Declarations

Conflicts of interest All the authors declare they have no conflict of interest.

8 Appendix

8.1 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. 8) taken from the artifact associated with the Optician papers [32, 34].

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 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 $\mathsf{String} \to S$), (2) core transformation (of type $S \to T$), and (3) printing (of type $T \to \mathsf{String}$), 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 $S = (\mathsf{NonEmpty\ String}, \mathsf{Bool})$ and $T = (\mathsf{String}, \mathsf{Bool}, [\mathsf{String}])$, where:

$$\textbf{type}\ \mathsf{NonEmpty}\ a = (a, [a]) \qquad \text{-- head-biased non-empty lists}$$

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

$$\begin{array}{l}
\textbf{data}\ \mathsf{LC} = \mathsf{LA} \mid \mathsf{LB} \mid \cdots \mid \mathsf{LZ} \\
\textbf{data}\ \mathsf{UC} = \mathsf{UA} \mid \mathsf{UB} \mid \cdots \mid \mathsf{UZ} \\
\textbf{data}\ \mathsf{C} = \mathsf{Lower\ LC} \mid \mathsf{Upper\ UC} \mid \mathsf{UnderScore} \mid \mathsf{Dot} \mid \mathsf{Hyphen} \\
\textbf{type}\ \mathsf{LocalFolder} = (\mathsf{C}, [\mathsf{C}]) \\
\textbf{type}\ \mathsf{Directory} = (\mathsf{Bool}, [\mathsf{LocalFolder}]) \\
\textbf{type}\ \mathsf{NonEmptyDirectory} = (\mathsf{Bool}, \mathsf{LocalFolder}, [\mathsf{LocalFolder}]) \\
\textbf{type}\ \mathsf{FileAndFolder} = (\mathsf{LocalFolder}, \mathsf{Directory})
\end{array}$$

```
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\n
           folder: /Users/amiltner/lens/tests/flashfill/"),
    createrex("tests/flashfill/extract-filename.txt",
           "file: extract-filename.txt\nfolder: tests/flashfill/")
  }
```

Fig. 8 `extr-fname.boom` for bijective-lens synthesis (excerpt)

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 $r*$ into either $rr*|\varepsilon$ or $r*r|\varepsilon$ nondeterministically [34], which eventually transforms $rr*$ (head-biased non-empty lists) into $r(r*r|\varepsilon) = rr*r|r$ (one-step expansions of last-biased nonempty lists). The core transformation involves no structural transformations after this expansion. However, this expansion of the 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 [34].

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:

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

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

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

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) insist 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 are 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 versus 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 programmability but also enables effective synthesis methods.

8.2 A concrete input and output for Q1

To demonstrate that SYNBIT is able to generate relatively large and complex (for automatic program synthesis) programs, we give here the input specification and synthesized output corresponding to Q1 in our experiments (Table 4 in Sect. 5.2.1). Figures 9, 10, 11 and 12 represent the input specification given to SYNBIT and Fig. 13 is the corresponding output. Here, we used the rose tree datatype Tree (Fig. 9) to express XML fragments. Note that the output involves non-trivial exit conditions and reconciliation functions.

```
data Tree a = N a [Tree a]   -- polymorphic tree type

data Lab = A [Char] [Char]   -- Attribute
         | T [Char]          -- Text
         | E [Char]          -- Element

q1 :: Tree Lab -> Tree Lab
q1 t = let N (E "book") ts = t
       in N (E "toc") (q1_section ts)

q1_section :: [Tree Lab] -> [Tree Lab]
q1_section l = case l of
   []                         -> []
   N (A a b) [] : rest        -> N (A a b) [] : q1_section rest
   N (E "title") title : rest -> N (E "title") title : q1_section rest
   N (E "section") xs : rest  -> N (E "section") (q1_section xs) : q1_section rest
   node:rest                  -> q1_section rest
```

Fig. 9 Input *get* program

```
N
  (E "book")
  [ N (E "title") [N (T "Data on the Web") []]
  , N (E "author") [N (T "Serge Abiteboul") []]
  , N (E "author") [N (T "Peter Buneman") []]
  , N (E "author") [N (T "Dan Suciu") []]
  , N
      (E "section")
      [ N (A "id" "intro") []
      , N (A "difficulty" "easy") []
      , N (E "title") [N (T "Introduction") []]
      , N (E "p") [N (T "Text ... ") []]
      , N
          (E "section")
          [ N (E "title") [N (T "Audience") []]
          , N (E "p") [N (T "Text ... ") []]]
      , N
          (E "section")
          [ N (E "title") [N (T "Web Data and the Two Cultures") []]
          , N (E "p") [N (T "Text ... ") []]
          , N
              (E "figure")
              [ N (A "height" "400") []
              , N (A "width" "400") []
              , N
                  (E "title")
                  [N (T "Traditional client/server architecture") []]
              , N (E "image") [N (A "source" "csarch.gif") []]]
          , N (E "p") [N (T "Text ...") []]]]
  , N
      (E "section")
      [ N (A "id" "syntax") []
      , N (A "difficulty" "medium") []
      , N (E "title") [N (T "A Syntax For Data") []]
      , N (E "p") [N (T "Text ... ") []]
      , N
          (E "figure")
          [ N (A "height" "200") []
          , N (A "width" "500") []
          , N (E "title") [N (T "Graph representations of structures") []]
          , N (E "image") [N (A "source" "graphs.gif") []]]
      , N (E "p") [N (T "Text ... ") []]
      , N
          (E "section")
          [ N (E "title") [N (T "Base Types") []]
          , N (E "p") [N (T "Text ...") []]]
      , N
          (E "section")
          [ N (E "title") [N (T "Representing Relational Databases") []]
          , N (E "p") [N (T "Text") []]
          , N
              (E "figure")
              [ N (A "height" "250") []
              , N (A "width" "400") []
              , N (E "title") [N (T "Examples of Relations") []]
              , N (E "image") [N (A "source" "relatios.gif") []]]]
      , N
          (E "section")
          [ N (E "title") [N (T "Representing Object Databases") []]
          , N (E "p") [N (T "Text ... ") []]]]]
```

Fig. 10 Original source

```
N
  (E "toc")
  [ N (E "title") [N (T "Data on the Web") []]
  , N
      (E "section")
      [ N (A "id" "intro") []
      , N (A "newattr" "attr") []  -- added
      , N (A "difficulty" "easy") []
      , N (E "title") [N (T "Introduction") []]
      , N (E "section") [N (E "title") [N (T "Audience") []]]]
      --   , N
      --       (E "section")
      --       [N (E "title") [N (T "Web Data and the Two Cultures") []]]
  , N
      (E "section")
      [ N (A "id" "syntax") []
      , N (A "difficulty" "hard") [] -- easy -> hard
      , N (E "title") [N (T "A Syntax For Data") []]
      , N (E "section") [N (E "title") [N (T "Base Types") []]]
      , N
          (E "section")
          [ N
              (E "title")
              [N (T "Representing Relational Databases and so on") []]]  -- changed title
      , N
          (E "section")
          [N (E "title") [N (T "Representing Object Databases") []]]
      , N (E "section") [N (E "title") [N (T "new section") []]]]  -- added
  , N (E "section") [N (E "title") [N (T "new section") []]]]      -- added
```

Fig. 11 Updated view (with comments added to denote the changes from an original view)

```
N
  (E "book")
  [ N (E "title") [N (T "Data on the Web") []]
  , N (E "author") [N (T "Serge Abiteboul") []]
  , N (E "author") [N (T "Peter Buneman") []]
  , N (E "author") [N (T "Dan Suciu") []]
  , N
      (E "section")
      [ N (A "id" "intro") []
      , N (A "newattr" "attr") []  -- added
      , N (A "difficulty" "easy") []
      , N (E "title") [N (T "Introduction") []]
      , N (E "p") [N (T "Text ... ") []]
      , N
          (E "section")
          [ N (E "title") [N (T "Audience") []]
          , N (E "p") [N (T "Text ... ") []]]]
          -- , N
          --     (E "section")
          --     [ N (E "title") [N (T "Web Data and the Two Cultures") []]
          --     , N (E "p") [N (T "Text ... ") []]
          --     , N
          --         (E "figure")
          --         [ N (A "height" "400") []
          --         , N (A "width" "400") []
          --         , N
          --             (E "title")
          --             [N (T "Traditional client/server architecture") []]
          --         , N (E "image") [N (A "source" "csarch.gif") []]]
          --     , N (E "p") [N (T "Text ...") []]]]
  , N
      (E "section")
      [ N (A "id" "syntax") []
      , N (A "difficulty" "hard") []  -- easy -> hard
      , N (E "title") [N (T "A Syntax For Data") []]
      , N (E "p") [N (T "Text ... ") []]
      , N
          (E "figure")
          [ N (A "height" "200") []
          , N (A "width" "500") []
          , N (E "title") [N (T "Graph representations of structures") []]
          , N (E "image") [N (A "source" "graphs.gif") []]]
      , N (E "p") [N (T "Text ... ") []]
      , N
          (E "section")
          [ N (E "title") [N (T "Base Types") []]
          , N (E "p") [N (T "Text ...") []]]
      , N
          (E "section")
          [ N
              (E "title")
              [N (T "Representing Relational Databases and so on") []]  -- changed title
          , N (E "p") [N (T "Text") []]
          , N
              (E "figure")
              [ N (A "height" "250") []
              , N (A "width" "400") []
              , N (E "title") [N (T "Examples of Relations") []]
              , N (E "image") [N (A "source" "relatios.gif") []]]]
      , N
          (E "section")
          [ N (E "title") [N (T "Representing Object Databases") []]
          , N (E "p") [N (T "Text ... ") []]]
      , N (E "section") [N (E "title") [N (T "new section") []]]]  -- added
  , N (E "section") [N (E "title") [N (T "new section") []]]]      -- added
```

Fig. 12 Updated source (with comments added to denote the changes from the original source)

```
q1 :: BX (Tree Lab) -> BX (Tree Lab)
q1 = \t -> case* t of
    N (E "book") ts ->
        N (E (| "toc" |)) (q1_section ts)
        with (\x1 -> True)
        reconciled by (\x0 -> \x1 -> x0)

q1_section :: BX [Tree Lab] -> BX [Tree Lab]
q1_section = \l -> case* l of
    [] ->
        ![]
        with (\x1 -> case x1 of
                        [] -> True
                        _ -> False)
        reconciled by (\x0 -> \x1 -> [])
    N (A a b) [] : rest ->
        (|N (A a b) ![] : q1_section rest|)
        with (\x1 -> case x1 of
                        N (A x2 x3) [] : x4 -> True
                        _ -> False)
        reconciled by (\x0 -> \x1 -> case x1 of
                                        N (A x2 x3) [] : x4 ->
                                            N (A x2 x2) [] : x0)
    N (E "title") title : rest ->
        (|N (E (| "title" |)) title : q1_section rest|)
        with (\x1 -> case x1 of
                        N (E "title") x2 : x3 -> True
                        _ -> False)
        reconciled by (\x0 -> \x1 -> case x1 of
                                        N (E "title") x2 : x3 -> x0)
    N (E "section") xs : rest ->
        (|N (E (| [ "section"  ] |)) (q1_section xs) : q1_section rest|)
        with (\x1 -> case x1 of
                        N (E "section") x2 : x3 -> True
                        _ -> False)
        reconciled by (\x0 -> \x1 -> case x1 of
                                        N (E "section") x2 : x3 ->
                                            N (E "section") x2 : x0)
    node : rest ->
        q1_section rest
        with (\x1 -> True)
        reconciled by (\x0 -> \x1 -> x0)
```

Fig. 13 Output of SYNBIT

8.3 A concrete input and output for lexer and parser

To demonstrate that SYNBIT is able to generate a simple recursive decent (specifically, LL(1)) lexer and parser, we give here the input *get* function and specification, and synthesized output corresponding to our experiments in Sect. 5.2.2. For lexer, Fig. 14, Table 7 and Fig. 15 represent the given *get*, the input specification and the corresponding output respectively. For parser, Fig. 16, Table 8 and Fig. 7 (in Sect. 5.2.2) represent the given *get*, the input specification and the corresponding output respectively.

```
tokenize :: [Char] → [Token]
tokenize cs = case cs of
    []                 → []
    '(' : cs'          → LPar : tokenize cs'
    ')' : cs'          → RPar : tokenize cs'
    '+' : cs'          → Plus : tokenize cs'
    'Z' : cs'          → TNum Z : tokenize cs'
    'S' : '(' : cs'    → let (n, ')' : cs'') = num cs' in
                           TNum (S n) : tokenize cs''

num :: [Char] → (Nat, [Char])
num cs = case cs of
    'Z' : cs'          → (Z, cs')
    'S' : '(' : cs'    → let (n, ')' : cs'') = num cs' in
                           (S n, cs'')
```

Fig. 14 An unidirectional lexer

Table 7 Input/output examples of lexer: for readability, we shall write v for [LPar, LPar, TNum (S (S Z)), RPar, Plus, LPar, TNum Z, RPar, RPar, Plus, LPar, TNum (S Z), RPar]

Original source	(Original view)	Updated view	Updated source
`"S(S(S(Z)))"`	[TNum (S (S (S Z)))]	[TNum (S (S Z))]	`"S(S(Z))"`
`"((S(S(Z)))+(Z))+(S(Z))"`	v	[TNum Z]	`"Z"`
`"Z"`	[TNum Z]	v	`"((S(S(Z)))+(Z))+(S(Z))"`
`"Z"`	[TNum Z]	[TNum (S (S Z))]	`"S(S(Z))"`

```
tokenize :: B[Char] → B[Token]
tokenize cs = case cs of
    []             → []
                     with λv. case v of {[] → True;  _ → False}
                     by λ_.λ_. []
    '(' : cs'      → LPar : tokenize cs'
                     with λv. case v of {LPar : _ → True;  _ → False}
                     by λs.λv. case v of {LPar : _ → '(' : s}
    ')' : cs'      → RPar : tokenize cs'
                     with λv. case v of {RPar : _ → True;  _ → False}
                     by λs.λv. case v of {RPar : _ → ')' : s}
    '+' : cs'      → Plus : tokenize cs'
                     with λv. case v of {Plus : _ → True;  _ → False}
                     by λs.λv. case v of {Plus : _ → '+' : s}
    'Z' : cs'      → TNum Z : tokenize cs'
                     with λv. case v of {TNum Z : _ → True;  _ → False}
                     by λs.λv. case v of {TNum Z : _ → 'Z' : s}
    'S' : '(' : cs' → let (n, ')' : cs'') = num cs' in
                     TNum (S n) : tokenize cs''
                     with λv. case v of {TNum (S _) : _ → True;  _ → False}
                     by λs.λv. case v of {TNum (S _) : _ → 'S' : '(' : 'Z' : ')' : s}

num :: [Char] → (Nat, [Char])
num cs = case cs of
    'Z' : cs'      → (Z, cs')
                     with λv. case v of {(Z, _) → True;  _ → False}
                     by λs.λv. case v of {(Z, _) → 'Z' : s}
    'S' : '(' : cs' → let (n, ')' : cs'') = num cs' in
                     (S n, cs'')
                     with λv. case v of {(S _, _) → True;  _ → False}
                     by λs.λv. case v of {(S _, _) → 'S' : '(' : s}
```

Fig. 15 Synthesized bidirectional lexer

```
pExp :: [Token] → Exp
pExp ts = let (e, []) = go ts in e

go :: [Token] → (Exp, [Token])
go ts = case ts of
  TNum n : r → (ENum n , r)
  LPar : r₁    → let (e₁, RPar : Plus : LPar : r₂) = go r₁ in
                 let (e₂, RPar : Plus : LPar : r₃) = go r₂ in
                 (EAdd e₁ e₂ , r₃)
```

Fig. 16 An unidirectional parser

Table 8 Input/output examples of parser: for readability, we shall write s for *tokenize* `"((S(S(Z)))+(Z))+(S(Z))"` and v for EAdd (EAdd (ENum (S (S Z))) (ENum Z)) (ENum (S Z))

Original source	(Original view)	Updated view	Updated source
s	v	[TNum (S (S Z))]	`"S(S(Z))"`
`"S(S(Z))"`	[TNum (S (S Z))]	v	s

References

1. Bancilhon F, Spyratos N (1981) Update semantics of relational views. ACM Trans Database Syst 6(4):557–575. https://doi.org/10.1145/319628.319634
2. Hegner SJ (1990) Foundations of canonical update support for closed database views. In: ICDT, pp 422–436. https://doi.org/10.1007/3-540-53507-1_93
3. Stevens P In: Lämmel R, Visser J, Saraiva J (eds) (2008) A landscape of bidirectional model transformations. Springer, Berlin, Heidelberg, pp 408–424. https://doi.org/10.1007/978-3-540-88643-3_10
4. Foster JN, Greenwald MB, Moore JT, Pierce BC, Schmitt A (2007) Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans Program Lang Syst 10(1145/1232420):1232424
5. Matsuda K, Hu Z, Nakano K, Hamana M, Takeichi M (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, pp 47–58. https://doi.org/10.1145/1291151.1291162
6. Bohannon A, Foster JN, Pierce BC, Pilkiewicz A, Schmitt A (2008) Boomerang: resourceful lenses for string data. In: Necula GC, Wadler P (eds) Proceedings of the 35th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008, pp 407–419 . https://doi.org/10.1145/1328438.1328487
7. Voigtländer J (2009) Bidirectionalization for free! (pearl). In: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages. POPL '09. Association for Computing Machinery, New York, NY, USA, pp 165–176. https://doi.org/10.1145/1480881.1480904
8. Pacheco H, Hu Z, Fischer S (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, pp 39–50. https://doi.org/10.1145/2543728.2543737
9. Matsuda K, Wang M (2018) Hobit: Programming lenses without using lens combinators. In: Ahmed A (ed) 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, pp 31–59. https://doi.org/10.1007/978-3-319-89884-1_2
10. Matsuda K, Wang M (2015) 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, pp 62–74. https://doi.org/10.1145/2784731.2784750
11. Matsuda K, Wang M (2015) "Bidirectionalization for free" for monomorphic transformations. Sci Comput Program 111:79–109. https://doi.org/10.1016/j.scico.2014.07.008
12. Chong N, Cook B, Kallas K, Khazem K, Monteiro FR, Schwartz-Narbonne D, Tasiran S, Tautschnig M, Tuttle MR (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, pp 11–20. https://doi.org/10.1145/3377813.3381347
13. Lubin J, Collins N, Omar C, Chugh R (2020) Program sketching with live bidirectional evaluation. In: Proceedings of the ACM on Programming Languages 4(ICFP), pp 109–110929. https://doi.org/10.1145/3408991
14. Gulwani S (2011) Automating string processing in spreadsheets using input-output examples. In: Ball T, Sagiv M (eds) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26–28, 2011, pp 317–330. https://doi.org/10.1145/1926385.1926423
15. Solar-Lezama A (2009) The sketching approach to program synthesis. In: Hu Z (ed) Programming languages and systems, 7th Asian symposium, APLAS 2009, Seoul, Korea, December 14–16, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5904, pp 4–13. https://doi.org/10.1007/978-3-642-10672-9_3
16. Lutz C (1986) Janus: a time-reversible language. Letter to R. Landauer. Available on: http://tetsuo.jp/ref/janus.pdf
17. Yokoyama T, Axelsen H.B, Glück R (2011) Towards a reversible functional language. In: RC, pp 14–29. https://doi.org/10.1007/978-3-642-29517-1_2
18. Ko H, Zan T, Hu Z (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, pp 61–72. https://doi.org/10.1145/2847538.2847544

19. Hu Z, Ko H (2016) Principles and practice of bidirectional programming in BiGUL. In: Bidirectional transformations—international summer school, Oxford, UK, July 25–29, 2016, Tutorial Lectures, pp 100–150. https://doi.org/10.1007/978-3-319-79108-1_4
20. Jha S, Gulwani S, Seshia SA, Tiwari A (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, pp 215–224. https://doi.org/10.1145/1806799.1806833
21. Feng Y, Martins R, Wang Y, Dillig I, Reps TW (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, pp 599–612. http://dl.acm.org/citation.cfm?id=3009851
22. Davies R, Pfenning F (2001) A modal analysis of staged computation. J ACM 48(3):555–604. https://doi.org/10.1145/382780.382785
23. Osera P, Zdancewic S (2015) Type-and-example-directed program synthesis. In: Grove D, Blackburn SM (eds) Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation, Portland, OR, USA, June 15–17, 2015, pp 619–630. https://doi.org/10.1145/2737924.2738007
24. Fischer S, Kiselyov O, Shan C (2011) Purely functional lazy nondeterministic programming. J Funct Program 21(4–5):413–465. https://doi.org/10.1017/S0956796811000189
25. Albarghouthi A, Gulwani S, Kincaid Z (2013) Recursive program synthesis. In: Computer aided verification—25th international conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings, pp 934–950. https://doi.org/10.1007/978-3-642-39799-8_67
26. Damas L, Milner R (1982) Principal type-schemes for functional programs. In: Conference record of the ninth annual ACM symposium on principles of programming languages, Albuquerque, New Mexico, USA, January 1982, pp 207–212. https://doi.org/10.1145/582153.582176
27. Matsuda K, Inaba K, Nakano K (2012) Polynomial-time inverse computation for accumulative functions with multiple data traversals. In: Proceedings of the ACM SIGPLAN 2012 workshop on partial evaluation and program manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23–24, 2012, pp 5–14. https://doi.org/10.1145/2103746.2103752
28. Nishida N, Vidal G (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, pp 283–298. https://doi.org/10.4230/LIPIcs.RTA.2011.283
29. de Jonge M, Visser E (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, pp 40–59. https://doi.org/10.1007/978-3-642-28830-2_3
30. Kort J, Lämmel R (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, p 161. https://doi.org/10.1109/SCAM.2003.1238042
31. Pombrio J, Krishnamurthi S (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, pp 361–371. https://doi.org/10.1145/2594291.2594319
32. Miltner A, Fisher K, Pierce BC, Walker D, Zdancewic S (2018) Synthesizing bijective lenses. In: Proceedings of the ACM on Programming Languages 2(POPL), pp 1–1130. https://doi.org/10.1145/3158089
33. Maina S, Miltner A, Fisher K, Pierce BC, Walker D, Zdancewic S (2018) Synthesizing quotient lenses. In: Proceedings of the ACM on Programming Languages 2(ICFP), pp 80–18029. https://doi.org/10.1145/3236775
34. Miltner A, Maina S, Fisher K, Pierce BC, Walker D, Zdancewic S (2019) Synthesizing symmetric lenses. In: Proceedings of the ACM on Programming Languages 3(ICFP), pp 95–19528. https://doi.org/10.1145/3341699
35. Hofmann M, Pierce BC, Wagner D (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, pp 371–384. https://doi.org/10.1145/1926385.1926428
36. Foster JN, Pilkiewicz A, Pierce BC (2008) Quotient lenses. In: Proceeding of the 13th ACM SIGPLAN international conference on functional programming, ICFP 2008, Victoria, BC, Canada, September 20–28, 2008, pp 383–396. https://doi.org/10.1145/1411204.1411257
37. Barbosa DMJ, Cretin J, Foster N, Greenberg M, Pierce BC (2010) Matching lenses: alignment and view update. In: Hudak P, Weirich S (eds) Proceeding of the 15th ACM SIGPLAN international conference on functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27–29, 2010, pp 193–204. ACM,. https://doi.org/10.1145/1863543.1863572
38. Voigtländer J (2012) Ideas for connecting inductive program synthesis and bidirectionalization. In: Proceedings of the ACM SIGPLAN 2012 workshop on partial evaluation and program manipulation,

PEPM 2012, Philadelphia, Pennsylvania, USA, January 23–24, 2012, pp 39–42. https://doi.org/10.1145/2103746.2103757
39. Srivastava S, Gulwani S, Chaudhuri S, Foster JS (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, pp 492–503. https://doi.org/10.1145/1993498.1993557
40. Korf RE (1981) Inversion of applicative programs. In: IJCAI, pp 1007–1009
41. Yokoyama T, Axelsen HB, Glück R (2008) Principles of a reversible programming language. In: Proceedings of the 5th conference on computing frontiers, 2008, Ischia, Italy, May 5–7, 2008, pp 43–54. https://doi.org/10.1145/1366230.1366239
42. Gries D (1981) The science of programming, chapter 21 inverting programs. Springer, Berlin
43. Glück R, Kawabe M (2005) Revisiting an automatic program inverter for lisp. SIGPLAN Not 40(5):8–17
44. Matsuda K, Mu S-C, Hu Z, Takeichi M (2010) A grammar-based approach to invertible programs. In: ESOP, pp 448–467
45. Nishida N, Sakai M, Sakabe T (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, pp 264–278. https://doi.org/10.1007/978-3-540-32033-3_20
46. Gomard CK, Jones ND (1991) A partial evaluator for the untyped lambda-calculus. J Funct Program 1(1):21–69. https://doi.org/10.1017/S0956796800000058
47. Almendros-Jiménez JM, Vidal G (2006) Automatic partial inversion of inductively sequential functions. In: Implementation and application of functional languages, 18th international symposium, IFL 2006, Budapest, Hungary, September 4–6, 2006, Revised Selected Papers, pp 253–270. https://doi.org/10.1007/978-3-540-74130-5_15
48. Matsuda K, Wang M (2020) Sparcl: a language for partially-invertible computation. In: Proceedings of the ACM on Programming Languages 4(ICFP), pp 118–111831. https://doi.org/10.1145/3409000
49. Wadler P (1990) Deforestation: transforming programs to eliminate trees. Theor Comput Sci 73(2):231–248. https://doi.org/10.1016/0304-3975(90)90147-A
50. Voigtländer J (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, pp 165–176. https://doi.org/10.1145/1480881.1480904
51. Wadler P (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, pp 347–359. https://doi.org/10.1145/99370.99404
52. Reynolds JC (1983) Types, abstraction and parametric polymorphism. In: Information processing 83, proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19–23, 1983, pp 513–523
53. Voigtländer J, Hu Z, Matsuda K, Wang M (2013) Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. J Funct Program 23(5):515–551. https://doi.org/10.1017/S0956796813000130
54. Matsuda K, Wang M (2018) Applicative bidirectional programming: Mixing lenses and semantic bidirectionalization. J Funct Program 28:15. https://doi.org/10.1017/S0956796818000096
55. Solar-Lezama A, Jones CG, Bodík R (2008) Sketching concurrent data structures. In: PLDI, pp 136–148. https://doi.org/10.1145/1375581.1375599
56. David C, Kesseli P, Kroening D, Lewis M (2018) Program synthesis for program analysis. ACM Trans Program Lang Syst 40(2):5–1545. https://doi.org/10.1145/3174802
57. David C, Kroening D, Lewis M (2015) Unrestricted termination and non-termination arguments for bit-vector programs. In: Vitek J (ed) Programming languages and systems—24th European symposium on programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11–18, 2015. Proceedings. Lecture Notes in Computer Science, vol. 9032, pp 183–204. https://doi.org/10.1007/978-3-662-46669-8_8
58. Abate A, Bessa I, Cattaruzza D, Cordeiro L.C, David C, Kesseli P, Kroening D, Polgreen E (2017) Automated formal synthesis of digital controllers for state-space physical plants. In: Majumdar R, Kuncak V (eds) Computer aided verification—29th international conference, CAV 2017, Heidelberg, Germany, July 24–28, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10426, pp 462–482. https://doi.org/10.1007/978-3-319-63387-9_23
59. Kneuss E, Kuraj I, Kuncak V, Suter P (2013) Synthesis modulo recursive functions. In: Hosking AL, Eugster PT, Lopes CV (eds) 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, pp 407–426. https://doi.org/10.1145/2509136.2509555
60. Abate A, David C, Kesseli P, Kroening D, Polgreen E (2018) Counterexample guided inductive synthesis modulo theories. In: Chockler H, Weissenbacher G (eds) 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, pp 270–288. https://doi.org/10.1007/978-3-319-96145-3_15

61. Katayama S (2005) Systematic search for lambda expressions. In: van Eekelen MCJD (ed) 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, pp 111–126
62. Feser JK, Chaudhuri S, Dillig I (2015) Synthesizing data structure transformations from input-output examples. In: Grove D, Blackburn SM (eds) Proceedings of the 36th ACM SIGPLAN conference on programming language design and implementation, Portland, OR, USA, June 15–17, 2015, pp 229–239. https://doi.org/10.1145/2737924.2737977
63. Smith C, Albarghouthi A (2019) Program synthesis with equivalence reduction. In: Enea C, Piskac R (eds) 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, pp 24–47. https://doi.org/10.1007/978-3-030-11245-5_2
64. Koukoutos M, Kneuss E, Kuncak V (2016) An update on deductive synthesis and repair in the leon tool. In: Piskac R, Dimitrova R (eds) Proceedings fifth workshop on synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016. EPTCS, vol. 229, pp 100–111. https://doi.org/10.4204/EPTCS.229.9