HOBiT: Programming Lenses Without Using Lens Combinators

Kazutaka Matsuda[1(✉)] and Meng Wang[2]

[1] Tohoku University, Sendai 980-8579, Japan
kztk@ecei.tohoku.ac.jp
[2] University of Bristol, Bristol BS8 1TH, UK

Abstract. We propose HOBiT, a higher-order bidirectional programming language, in which users can write bidirectional programs in the familiar style of conventional functional programming, while enjoying the full expressiveness of lenses. A bidirectional transformation, or a lens, is a pair of mappings between source and view data objects, one in each direction. When the view is modified, the source is updated accordingly with respect to some laws—a pattern that is found in databases, model-driven development, compiler construction, and so on. The most common way of programming lenses is with lens combinators, which are lens-to-lens functions that compose simpler lenses to form more complex ones. Lens combinators preserve the bidirectionality of lenses and are expressive; but they compel programmers to a specialised point-free style—i.e., no naming of intermediate computation results—limiting the scalability of bidirectional programming. To address this issue, we propose a new bidirectional programming language HOBiT, in which lenses are represented as standard functions, and combinators are mapped to language constructs with binders. This design transforms bidirectional programming, enabling programmers to write bidirectional programs in a flexible functional style and at the same time access the full expressiveness of lenses. We formally define the syntax, type system, and the semantics of the language, and then show that programs in HOBiT satisfy bidirectionality. Additionally, we demonstrate HOBiT's programmability with examples.

1 Introduction

Transforming data from one format to another is a common task of programming: compilers transform program texts into syntax trees, manipulate the trees and then generate low-level code; database queries transform base relations into views; model transformations generate lower-level implementations from higher-level models; and so on. Very often, such transformations will benefit from being bidirectional, allowing changes to the targets to be mapped back to the sources too. For example, if one can run a compiler front-end (preprocessing, parsing, desugaring, etc.) backwards, then all sorts of program analysis tools will be able to focus on a much smaller core language, without sacrificing usability, as

A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 31–59, 2018.
https://doi.org/10.1007/978-3-319-89884-1_2

their outputs in term of the core language will be transformed backwards to the source language. In the same way, such needs arise in databases (the *view-update problem* [1,6,12]) and model-driven engineering (bidirectional model transformation) [28,33,35].

As a response to this challenge, programming language researchers have started to design languages that execute deterministically in both directions, and the lens framework is the most prominent among all. In the lens framework, a *bidirectional transformation* (or a *lens*) $\ell \in \mathit{Lens}\ S\ V$, consists of $\mathit{get}\ \ell \in S \to V$, and $\mathit{put}\ \ell \in S \to V \to S$ [3,7,8]. (When clear from the context, or unimportant, we sometimes omit the lens name and write simply *get*/*put*.) Function *get* extracts a view from a source, and *put* takes both an updated view and the original source as inputs to produce an updated source. The additional parameter of *put* makes it possible to recover some of the source data that is not present in the view. In other words, *get* needs not to be injective to have a *put*. Not all pairs of *get*/*put* are considered correct lenses. The following round-triping laws of a lens ℓ are generally required to establish bidirectionality:

$$\mathit{put}\ \ell\ s\ v = s \quad \text{if} \quad \mathit{get}\ \ell\ s = v \qquad \textbf{(Acceptability)}$$
$$\mathit{get}\ \ell\ s' = v \quad \text{if} \quad \mathit{put}\ \ell\ s\ v = s' \qquad \textbf{(Consistency)}$$

for all s, s' and v. (In this paper we write $e = e'$ with the assumption that neither e nor e' is undefined. Stronger variants of the laws enforcing totality exist elsewhere, for example in [7].) Here *consistency* ensures that all updates on a view are captured by the updated source, and *acceptability* prohibits changes to the source if no update has been made on the view. Collectively, the two laws defines *well-behavedness* [1,7,12].

The most common way of programming lenses is with lens combinators [3,7,8], which are basically a selection of lens-to-lens functions that compose simpler lenses to form more complex ones. This combinator-based approach follows the long history of lightweight language development in functional programming. The distinctive advantage of this approach is that by restricting the lens language to a few selected combinators, well-behavedness can be more easily preserved in programming, and therefore given well-behaved lenses as inputs, the combinators are guaranteed to produce well-behaved lenses. This idea of lens combinators is very influential academically, and various designs and implementations have been proposed [2,3,7–9,16,17,27,32] over the years.

1.1 The Challenge of Programmability

The complexity of a piece of software can be classified as either intrinsic or accidental. Intrinsic complexity reflects the inherent difficulty of the problem at hand, whereas accidental complexity arises from the particular programming language, design or tools used to implement the solution. This work aims at reducing the accidental complexity of bidirectional programming by contributing to the design of bidirectional languages. In particularly, we identify a language restriction—i.e., no naming of intermediate computation results—which complicates lens programming, and propose a new design that removes it.

As a teaser to demonstrate the problem, let us consider the list append function. In standard unidirectional programming, it can be defined simply as $append\ x\ y = \mathbf{case}\ x\ \mathbf{of}\ \{[\,] \to y;\ a : x' \to a : append\ x'\ y\}$. Astute readers may have already noticed that *append* is defined by structural recursion on x, which can be made explicit by using *foldr* as in $append\ x\ y = foldr\ (:)\ y\ x$.

But in a lens language based on combinators, things are more difficult. Specifically, *append* now requires a more complicated recursion pattern, as below.

```
appendL :: Lens ([A], [A]) [A]
appendL =
  cond idL (λ_.True) (λ_.λ_.[]) (consL ◦̂ (idL × appendL)) (not ∘ null) (λ_.λ_.⊥)
  ◦̂ rearr ◦̂ (outListL × idL)
 where outListL :: Lens [A] (Either ( ) (A, [A]))
       rearr    :: Lens (Either ( ) (a, b), c) (Either c (a, (b, c)))
       (◦̂)      :: Lens b c → Lens a b → Lens a c
       cond     :: Lens a c → ... → Lens b c → ... → Lens (Either a b) c
       ...
```

It is beyond the scope of this paper to explain how exactly the definition of *appendL* works, as its obscurity is what this work aims to remove. Instead, we informally describe its behaviour and the various components of the code. The above code defines a lens: forwards, it behaves as the standard *append*, and backwards, it splits the updated view list, and when the length of the list changes, this definition implements (with the grayed part) the bias of keeping the length of the first source list whenever possible (to disambiguate multiple candidate source changes). Here, *cond*, $(\hat{\circ})$, etc. are lens combinators and *outListL* and *rearr* are auxiliary lenses, as can be seen from their types. Unlike its unidirectional counterpart, *appendL* can no longer be defined as a structural recursion on list; instead it traverses a pair of lists with rather complex rearrangement *rearr*.

Intuitively, the additional grayed parts is intrinsic complexity, as they are needed for directing backwards execution. However, the complicated recursion scheme, which is a direct result of the underlying limitation of lens languages, is certainly accidental. Recall that in the definition of *append*, we were able to use the variable y, which is bound outside of the recursion pattern, inside the body of *foldr*. But the same is not possible with lens combinators which are strictly 'pointfree'. Moreover, even if one could name such variables (points), their usage with lens combinators will be very restricted in order to guarantee well-behavedness [21,23]. This problem is specific to opaque non-function objects such as lenses, and goes well beyond the traditional issues associated with the pointfree programming style.

In this paper, we design a new bidirectional language HOBiT, which aims to remove much of the accidental difficulty found in combinator-based lens programming, and reduces the gap between bidirectional programming and standard functional programming. For example, the following definition in HOBiT implements the same lens as *appendL*.

$$
\begin{aligned}
&\mathit{appendB} :: \mathbf{B}[A] \to \mathbf{B}[A] \to \mathbf{B}[A] \\
&\mathit{appendB}\ x\ y = \underline{\mathbf{case}}\ x\ \underline{\mathbf{of}}\ [\,] \quad \to y \qquad\qquad\qquad\qquad \underline{\mathbf{with}}\ \lambda\_.\mathsf{True}\ \ \underline{\mathbf{by}}\ (\lambda\_.\lambda\_.[\,]) \\
&\qquad\qquad\qquad\qquad\quad a : x' \to a \mathbin{\underline{:}} \mathit{appendB}\ x'\ y\ \underline{\mathbf{with}}\ \mathit{not} \circ \mathit{null}\ \underline{\mathbf{by}}\ (\lambda\_.\lambda\_.\bot)
\end{aligned}
$$

As expected, the above code shares the grayed part with the definition of *appendL* as the two implement the same backwards behaviour. The difference is that *appendB* uses structural recursion in the same way as the standard unidirectional *append*, greatly simplifying programming. This is made possible by the HOBiT's type system and semantics, allowing unrestricted use of free variables. This difference in approach is also reflected in the types: *appendB* is a proper function (instead of the abstract lens type of *appendL*), which readily lends itself to conventional functional programming. At the same time, *appendB* is also a proper lens, which when executed by the HOBiT interpreter behave exactly like *appendL*. A major technical challenge in the design of HOBiT is to guarantee this duality, so that functions like *appendB* are well-behaved by construction despite the flexibility in their construction.

1.2 Contributions

As we can already see from the very simple example above, the use of HOBiT simplifies bidirectional programming by removing much of the accidental complexity. Specifically, HOBiT stands out from existing bidirectional languages in two ways:

1. It supports the conventional programming style that is used in unidirectional programming. As a result, a program in HOBiT can be defined in a way similar to how one would define only its *get* component. For example, *appendB* is defined in the same way as the unidirectional *append*.
2. It supports incremental improvement. Given the very often close resemblance of a bidirectional-program definition and that of its *get* component, it becomes possible to write an initial version of a bidirectional program almost identical to its *get* component and then to adjust the backwards behaviour gradually, without having to significantly restructure the existing definition.

Thanks to these distinctive advantages, HOBiT for the first time allows us to construct realistically-sized bidirectional programs with relative ease. Of course, this does not mean free lunch: the ability to control backwards behaviours will not magically come without additional code (for example the grayed part above). What HOBiT achieves is that programming effort may now focus on the productive part of specifying backwards behaviours, instead of being consumed by circumventing language restrictions.

In summary, we make the following contributions in this paper.

– We design a higher-order bidirectional programming language HOBiT, which supports convenient bidirectional programming with control of backwards behaviours (Sect. 3). We also discuss several extensions to the language (Sect. 5).

– We present the semantics of HOBiT inspired by the idea of staging [5], and prove the well-behavedness property using Kripke logical relations [18] (Sect. 4).
– We demonstrate the programmability of HOBiT with examples such as desugaring/resugaring [26] (Sect. 6). Additional examples including a bidirectional evaluator for λ-calculus [21,23], a parser/printer for S-expressions, and bookmark extraction for Netscape [7] can be found at https://bitbucket.org/kztk/hibx together with a prototype implementation of HOBiT.

2 Overview: Bidirectional Programming Without Combinators

In this section, we informally introduce the essential constructs of HOBiT and demonstrate their use by a few small examples. Recall that, as seen in the *appendB* example, the strength of HOBiT lies in allowing programmers to access λ-abstractions without restrictions on the use of λ-bound variables.

2.1 The case Construct

The most important language construct in HOBiT is **case** (pronounced as *bidirectional case*), which provides pattern matching and easy access to bidirectional branching, and also importantly, allows unrestricted use of λ-bound variables.

In general, a **case** expression has the following form.

$$\textbf{case}\ e\ \textbf{of}\ \{p_1 \to e_1\ \textbf{with}\ \phi_1\ \textbf{by}\ \rho_1; \ldots; p_n \to e_n\ \textbf{with}\ \phi_n\ \textbf{by}\ \rho_n\}$$

(Like Haskell, we shall omit "{", "}" and ";" if they are clear from the layout.) In the type system of HOBiT, a **case**-expression has type $\mathbf{B}B$, if e and e_i have types $\mathbf{B}A$ and $\mathbf{B}B$, and ϕ_i and ρ_i have types $B \to \mathit{Bool}$ and $A \to B \to A$, where A and B contains neither $(\to)$ nor $\mathbf{B}$. The type $\mathbf{B}A$ can be understood intuitively as "updatable A". Typically, the source and view data are given such $\mathbf{B}$-types, and a function of type $\mathbf{B}A \to \mathbf{B}B$ is the HOBiT equivalent of *Lens A B*.

The pattern matching part of **case** performs two implicit operations: it first unwraps the $\mathbf{B}$-typed value, exposing its content for normal pattern matching, and then it wraps the variables bound by the pattern matching, turning them into 'updatable' $\mathbf{B}$-typed values to be used in the bodies. For example, in the second branch of *appendB*, a and x' can be seen as having types A and $[A]$ in the pattern, but $\mathbf{B}A$ and $\mathbf{B}[A]$ types in the body; and the bidirectional constructor $(\underline{:}) :: \mathbf{B}A \to \mathbf{B}[A] \to \mathbf{B}[A]$ combines them to produce a $\mathbf{B}$-typed list.

In addition to the standard conditional branches, **case**-expression has two unique components ϕ_i and ρ_i called *exit conditions* and *reconciliation functions* respectively, which are used in backwards executions. Exit condition ϕ_i is an over-approximation of the forwards-execution results of the expressions e_i. In other words, if branch i is choosen, then $\phi_i\ e_i$ must evaluate to True. This assertion is checked dynamically in HOBiT, though could be checked statically with

a sophisticated type system [7]. In the backwards direction the exit condition is used for deciding branching: the branch with its exit condition satisfied by the updated view (when more than one match, the original branch used in the forwards direction has higher priority) will be picked for execution. The idea is that due to the update in the view, the branch taken in the backwards direction may be different from the one taken in the original forwards execution, a feature that is commonly supported by lens languages [7] which we call *branch switching*.

Branch switching is crucial to *put*'s *robustness*, i.e., the ability to handle a wide range of view updates (including those affect the branching decisions) without failing. We explain its working in details in the following.

Branch Switching. Being able to choose a different branch in the backwards direction only solves part of the problem. Let us consider the case where a forward execution chooses the n^{th} branch, and the backwards execution, based on the updated view, chooses the m^{th} ($m \neq n$) branch. In this case, the original value of the pattern-matched expression e, which is the reason for the n^{th} branch being chosen, is not compatible with the *put* of the m^{th} branch.

As an example, let us consider a simple function that pattern-matches on an *Either* structure and returns an list. Note that we have purposely omitted the reconciliation functions.

```
f :: B(Either [A] (A, [A])) → B[A]
f x = case x of Left ys       → ys      with λ_.True  {- no by here -}
                Right (y, ys) → y : ys  with not ∘ null
```

We have said that functions of type $\mathbf{B}A \to \mathbf{B}B$ are also fully functioning lenses of type *Lens A B*. In HOBiT, the above code runs as follows, where `HOBiT>` is the prompt of HOBiT's read-eval-print loop, and `:get` and `:put` are meta-language operations to perform *get* and *put* respectively.

```
HOBiT> :get f (Left [1, 2, 3])
[1, 2, 3]
HOBiT> :get f (Right (1, [2, 3]))
[1, 2, 3]
HOBiT> :put f (Left [1, 2, 3]) [4, 5]     -- The view [1, 2, 3] is updated to [4, 5].
Left [4, 5]                               -- Both exit conditions are true with [4, 5],
                                          -- so the original branch (Left) is taken.
HOBiT> :put f (Right (1, [2, 3])) [4, 5]
Right (4, [5])                            -- Similar, but the original branch is Right.
HOBiT> :put f (Right (1, [2, 3])) []
⊥                                         -- Branch switches, but computation fails.
```

As we have explained above, exit conditions are used to decide which branch will be used in the backwards direction. For the first and second evaluations of *put*, the exit conditions corresponding to the original branches were true for the updated view. For the last evaluation of *put*, since the exit condition of

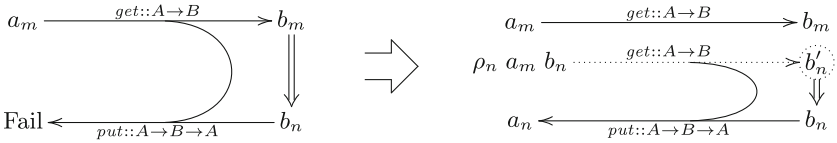

Fig. 1. Reconciliation function: assuming exit conditions ϕ_m and ϕ_n where $\phi_m\ b_n =$ False but $\phi_n\ b_n =$ True, and reconciliation functions ρ_m and ρ_n.

the original branch was false but that of the other branch was true, branch switching is required here. However, a direct *put*-execution of f with the inputs (Right $(1, [2, 3])$) and $[\,]$ crashes (represented by $\bot$ above), for a good reason, as the two inputs are in an inconsistent state with respect to f.

This is where reconciliation functions come into the picture. For the Left branch above, a sensible reconciliation function will be $(\lambda\_.\lambda\_.\mathsf{Left}\ [\,])$, which when applied turns the conflicting source (Right $(1, [2, 3])$) into Left $[\,]$, and consequently the *put*-execution may succeed with the new inputs and returns Left $[\,]$. It is not difficult to verify that the "reconciled" *put*-execution still satisfies well-behavedness. Note that despite the similarity in types, reconciliation functions are not *put*; they merely provide a default source value to allow stuck *put*-executions to proceed. We visualise the effect of reconciliation functions in Fig. 1. The left-hand side is bidirectional execution without successful branch-switching, and since $\phi_m\ b_n$ is false (indicating that b_n is not in the range of the m^{th} branch) the execution of *put* must (rightfully) fail in order to guarantee well-behavedness. On the right-hand side, reconciliation function ρ_n produces a suitable source from a_m and b_n (where $\phi_n\ (get\ (\rho_n\ a_m\ b_n))$ is True), and *put* executes with b_n and the new source $\rho_n\ a_m\ b_n$. It is worth mentioning that branch switching with reconciliation functions does not compromise correctness: though the quality of the user-defined reconciliation functions affects robustness as they may or may not be able to resolve conflicts, successful *put*-executions always guarantee well-behavedness, regardless the involvement of reconciliation functions.

Revisiting *appendB*. Recall *appendB* from Sect. 1.1 (reproduced below).

$$
\begin{aligned}
&\mathit{appendB} :: \mathbf{B}[A] \to \mathbf{B}[A] \to \mathbf{B}[A] \\
&\mathit{appendB}\ x\ y = \underline{\mathbf{case}}\ x\ \underline{\mathbf{of}}\ [\,] \quad \to y \qquad\qquad\qquad\qquad \underline{\mathbf{with}}\ \lambda\_.\mathsf{True} \quad \underline{\mathbf{by}}\ (\lambda\_.\lambda\_.[\,]) \\
&\qquad\qquad\qquad\qquad\quad a : x' \to a : \mathit{appendB}\ x'\ y\ \underline{\mathbf{with}}\ \mathit{not} \circ \mathit{null}\ \underline{\mathbf{by}}\ (\lambda\_.\lambda\_.\bot)
\end{aligned}
$$

The exit condition for the nil case always returns true as there is no restriction on the value of y, and for the cons case it requires the returned list to be non-empty. In the backwards direction, when the updated view is non-empty, both exit conditions will be true, and then the original branch will be taken. This means that since *appendB* is defined as a recursion on x, the backwards execution will try to unroll the original recursion step by step (i.e., the cons branch will be taken for a number of times that is the same as the length of x) as long as the view remains non-empty. If an updated view list is shorter than x, then $\mathit{not} \circ \mathit{null}$

will become false before the unrolling finishes, and the nil branch will be taken (branch-switching) and the reconciliation function will be called.

The definition of *appendB* is curried; straightforward uncurrying turns it into the standard form $\mathbf{B}A \to \mathbf{B}B$ that can be interpreted by HOBiT as a lens. The following HOBiT program is the bidirectional variant of *uncurry*.

$$\begin{aligned}&\mathit{uncurryB} :: (\mathbf{B}A \to \mathbf{B}B \to \mathbf{B}C) \to \mathbf{B}(A, B) \to \mathbf{B}C\\ &\mathit{uncurryB}\ f\ z = \underline{\mathbf{let}}\ (x, y) = z\ \underline{\mathbf{in}}\ f\ x\ y\end{aligned}$$

Here, $\underline{\mathbf{let}}\ p = e\ \underline{\mathbf{in}}\ e'$ is syntactic sugar for $\underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p \to e'\ \mathbf{with}\ (\lambda\_.\mathsf{True})\ \underline{\mathbf{by}}\ (\lambda s.\lambda\_.s)\}$, in which the reconciliation function is never called as there is only one branch. Let $\mathit{appendB}' = \mathit{uncurryB}\ \mathit{appendB}$, then we can run $\mathit{appendB}'$ as:

```
HOBiT> :get appendB' ([1,2],[3,4,5])
[1,2,3,4,5]
HOBiT> :put appendB' ([1,2],[3,4,5]) [6,7,8,9,10]
([6,7],[8,9,10])      -- No structural change, no branch switching.
HOBiT> :put appendB' ([1,2],[3,4,5]) [6,7]
([6,7],[])      -- No branch switching, still.
HOBiT> :put appendB' ([1,2],[3,4,5]) [6]
([6],[])      -- Branch-switching happens and the recursion terminates early.
```

Difference from Lens Combinators. As mentioned above, the idea of branch switching can be traced back to lens languages. In particular, the design of **case** is inspired by the combinator *cond* [7]. Despite the similarities, it is important to recognise that **case** is not only a more convenient syntax for *cond*, but also crucially supports the unrestricted use of λ-bound variables. This more fundamental difference is the reason why we could define *appendB* in the conventional functional style as the variables x and y are used freely in the body of **case**. In other words, the novelty of HOBiT is its ability to combine the traditional (higher-order) functional programming and the bidirectional constructs as found in lens combinators, effectively establishing a new way of bidirectional programming.

2.2 A More Elaborate Example: *linesB*

In addition to supporting convenient programming and robustness in *put* execution, the **case** constructs can also be used to express intricate details of backwards behaviours. Let us consider the *lines* function in Haskell as an example, which splits a string into a list of strings by newlines, for example, *lines* `"AA\nBB\n"` = [`"AA"`, `"BB"`], except that the last newline character in its input is optional. For example, *lines* returns [`"AA"`, `"BB"`] for both `"AA\nBB\n"` and `"AA\nBB"`. Suppose that we want the backwards transformation of *lines* to exhibit a behaviour that depends on the original source:

```
linesB :: B String → B[String]
linesB str =
  let (f, b) = breakNLB str
  in case b of '\n' : x : r → f ⋮ linesB (x ⋮ r)
                                  with (> 1) ∘ length by (λb.λ_.'\n' : ' ' : b)
               b'           → f ⋮ [] with (== 1) ∘ length by (λb.λ_.lastNL b)
  where {lastNL [] = []; lastNL ['\n'] = ['\n']; lastNL (a : x) = lastNL x}
breakNLB :: B String → B(String, String)
breakNLB str = case str of
  []      → ([], [])                    with p1 by (λ_.λ_.[])
  '\n' : s → ([], '\n' ⋮ s)             with p2 by (λ_.λ_."\n")
  c : s   → let (f, r) = breakNLB s in (c ⋮ f, r) with p3 by (λ_.λ_." ")
  where {p1(x, y) = null y; p2(x, y) = null x && not (null y); p3(x, y) = not (null x)}
```

Fig. 2. *linesB* and *breakNLB*

```
HOBiT> :put linesB "AA\nBB" ["a", "b"]
"a\nb"
HOBiT> :put linesB "AA\nBB" ["a", "b", "c"]
"a\nb\nc"
HOBiT> :put linesB "AA\nBB" ["a"]
"a"
HOBiT> :put linesB "AA\nBB\n" ["a", "b", "c"]
"a\nb\nc\n"
HOBiT> :put linesB "AA\nBB\n" ["a"]
"a\n"
```

This behaviour is achieved by the definition in Fig. 2, which makes good use of reconciliation functions. Note that we do not consider the contrived corner case where the string ends with duplicated newlines such as in `"A\n\n"`. The function *breakNLB* splits a string at the first newline; since *breakNLB* is injective, its exit conditions and reconciliation functions are of little interest. The interesting part is in the definition of *linesB*, particularly its use of reconciliation functions to track the existence of a last newline character. We firstly explain the branching structure of the program. On the top level, when the first line is removed from the input, the remaining string b may contain more lines, or be the end (represented by either the empty list or the singleton list `['\n']`). If the first branch is taken, the returned result will be a list of more than one element. In the second branch when it is the end of the text, b could contain a newline or simply be empty. We do not explicitly give patterns for the two cases as they have the same body $f \mathbin{⋮} []$, but the reconciliation function distinguishes the two in order to preserve the original source structure in the backwards execution. Note that we intentionally use the same variable name b in the case analysis and the reconciliation function, to signify that the two represent the same source data. The use of argument b in the reconciliation functions serves the purpose of remembering the (non)existence of the last newline in the original source, which is then preserved in the new source.

$$
\begin{aligned}
e &::= x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{True} \mid \mathsf{False} \mid [\,] \mid e_1 : e_2 \mid \mathbf{case}\ e\ \mathbf{of}\ \{p_i \to e_i\}_{i=1,2} \mid \mathbf{fix}\ (\lambda f.e) \\
&\quad \mid \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid e_1 \mathbin{\underline{:}} e_2 \mid \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ e_i'\ \underline{\mathbf{by}}\ e_i''\}_{i=1,2} \\
p &::= x \mid \mathsf{True} \mid \mathsf{False} \mid [\,] \mid p_1 : p_2
\end{aligned}
$$

Fig. 3. Syntax of HOBiT Core

It is worth noting that just like the other examples we have seen, this definition in HOBiT shares a similar structure with a definition of *lines* in Haskell.[1] The notable difference is that a Haskell definition is likely to have a different grouping of the three cases of *lines* into two branches, as there is no need to keep track of the last newline for backwards execution. Recall that reconciliation functions are called *after* branches are chosen by exit conditions; in the case of *linesB*, the reconciliation function is used to decide the reconciled value of b' to be `"\n"` or `""`. This, however, means that we cannot separate the pattern b' into two `"\n"` and `""` with copying its branch body and exit condition, because then we lose a chance to choose a reconciled value of b based on its original value.

3 Syntax and Type System of HOBiT Core

In this section, we describe the syntax and the type system of the core of HOBiT.

3.1 Syntax

The syntax of HOBiT Core is given in Fig. 3. For simplicity, we only consider booleans and lists. The syntax is almost the same as the standard λ-calculus with the fixed-point combinator (**fix**), lists and booleans. For data constructors and case expressions, there are in addition bidirectional versions that are underlined. We allow the body of **fix** to be non-λs to make our semantics simple (Sect. 4), though such a definition like $\mathbf{fix}(\lambda x.\mathsf{True} : x)$ can diverge.

Although in examples we used **case**/**case**-expressions with an arbitrary number of branches having overlapping patterns under the first-match principle, we assume for simplicity that in HOBiT Core **case**/**case**-expressions must have exactly two branches whose patterns do not overlap; extensions to support these features are straightforward. As in Haskell, we sometimes omit the braces and semicolons if they are clear from the layout.

[1] Haskell's *lines*'s behaviour is a bit more complicated as it returns [] if and only if the input is `""`. This behaviour can be achieved by calling *linesB* only when the input list is nonempty.

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

$$\frac{\Gamma(x) = A}{\Gamma;\Delta \vdash x : A} \quad \frac{\Delta(x) = \sigma}{\Gamma;\Delta \vdash x : \mathbf{B}\sigma} \quad \frac{\Gamma, x : A;\Delta \vdash e : B}{\Gamma;\Delta \vdash \lambda x.e : A \to B} \quad \frac{\Gamma;\Delta \vdash e_1 : A \to B \quad \Gamma;\Delta \vdash e_2 : A}{\Gamma;\Delta \vdash e_1\ e_2 : B}$$

$$\frac{\Gamma, f : A;\Delta \vdash e : A}{\Gamma;\Delta \vdash \mathbf{fix}(\lambda f.e) : A} \quad \frac{}{\Gamma;\Delta \vdash \mathsf{True} : \mathit{Bool}} \quad \frac{}{\Gamma;\Delta \vdash \mathsf{False} : \mathit{Bool}} \quad \frac{}{\Gamma;\Delta \vdash [\,] : [A]}$$

$$\frac{\Gamma;\Delta \vdash e_1 : A \quad \Gamma;\Delta \vdash e_2 : [A]}{\Gamma;\Delta \vdash e_1 : e_2 : [A]} \quad \frac{}{\Gamma;\Delta \vdash \underline{\mathsf{True}} : \mathbf{B}\mathit{Bool}} \quad \frac{}{\Gamma;\Delta \vdash \underline{\mathsf{False}} : \mathbf{B}\mathit{Bool}} \quad \frac{}{\Gamma;\Delta \vdash \underline{[\,]} : \mathbf{B}[\sigma]}$$

$$\frac{\Gamma;\Delta \vdash e_1 : \mathbf{B}\sigma \quad \Gamma;\Delta \vdash e_2 : \mathbf{B}[\sigma]}{\Gamma;\Delta \vdash e_1 \mathbin{\underline{:}} e_2 : \mathbf{B}[\sigma]} \quad \frac{\Gamma;\Delta \vdash e : A \quad \Gamma_i \vdash p_i : A \quad \Gamma, \Gamma_i;\Delta \vdash e_i : B \quad (i = 1, 2)}{\Gamma;\Delta \vdash \mathbf{case}\ e\ \mathbf{of}\ \{p_i \to e_i\}_{i=1,2} : B}$$

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash e : \mathbf{B}\sigma \quad \Delta_i \vdash p_i : \sigma \quad \Gamma;\Delta, \Delta_i \vdash e_i : \mathbf{B}\tau \\ \Gamma;\Delta \vdash e'_i : \tau \to \mathit{Bool} \quad \Gamma;\Delta \vdash e''_i : \sigma \to \tau \to \sigma \quad (i = 1, 2)\end{array}}{\Gamma;\Delta \vdash \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ e'_i\ \underline{\mathbf{by}}\ e''_i\}_{i=1,2} : \mathbf{B}\tau}$$

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

$$\frac{}{x : A \vdash x : A} \quad \frac{}{\emptyset \vdash \mathsf{True} : \mathit{Bool}} \quad \frac{}{\emptyset \vdash \mathsf{False} : \mathit{Bool}} \quad \frac{}{\emptyset \vdash [\,] : [A]} \quad \frac{\Gamma_1 \vdash e_1 : A \quad \Gamma_2 \vdash e_2 : [A]}{\Gamma_1, \Gamma_2 \vdash e_1 : e_2 : [A]}$$

Fig. 4. Typing rules: $\Delta \vdash p : \sigma$ is similar to $\Gamma \vdash p : A$ but asserts that the resulting environment is actually a bidirectional environment.

3.2 Type System

The types in HOBiT Core are defined as follows.

$$A, B ::= \mathbf{B}\sigma \mid A \to B \mid [A] \mid \mathit{Bool}$$

We use the metavariable $\sigma, \tau, \ldots$ for types that do not contain $\to$ nor $\mathbf{B}$, We call σ-types *pure datatypes*, which are used for sources and views of lenses. Intuitively, $\mathbf{B}\sigma$ represents "updatable σ"—data subject to update in bidirectional transformation. We keep the type system of HOBiT Core simple, though it is possible to include polymorphic types or intersection types to unify unidirectional and bidirectional constructors.

The typing judgment $\Gamma;\Delta \vdash e : A$, which reads that under environments Γ and Δ, expression e has type A, is defined by the typing rules in Fig. 4. We use two environments: Δ (the *bidirectional type environment*) is for variables introduced by pattern-matching through **case**, and Γ for everything else. It is interesting to observe that Δ only holds pure datatypes, as the pattern variables of **case** have pure datatypes, while Γ holds any types. We assume that the variables in Γ and those in Δ are disjoint, and appropriate α-renaming has been done to ensure this. This separation of Δ from Γ does not affect typeability, but is key to our semantics and correctness proof (Sect. 4). Most of the rules are standard except **case**; recall that we only use unidirectional constructors in patterns which have pure types, while the variables bound in the patterns are used as $\mathbf{B}$-typed values in branch bodies.

4 Semantics of HOBiT Core

Recall that the unique strength of HOBiT is its ability to mix higher-order unidirectional programming with bidirectional programming. A consequence of this mixture is that we can no longer specify its semantics in the same way as other *first-order* bidirectional languages such as [13], where two semantics—one for *get* and the other for *put*—suffice. This is because the category of lenses is believed to have no exponential objects [27] (and thus does not permit λs).

4.1 Basic Idea: Staging

Our solution to this problem is staging [5], which separates evaluation into two stages: the unidirectional parts is evaluated first to make way for a bidirectional semantics, which only has to deal with the residual first-order programs. As a simple example, consider the expression $(\lambda z.z)\ (x \mathbin{\underline{:}} ((\lambda w.w)\ y) \mathbin{\underline{:}} \underline{[]})$. The first-stage evaluation, $e \Downarrow_{\mathrm{U}} E$, eliminates λs from the expression as in $(\lambda z.z)\ (x \mathbin{\underline{:}} ((\lambda w.w)\ y) \mathbin{\underline{:}} \underline{[]}) \Downarrow_{\mathrm{U}} x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[]}$. Then, our bidirectional semantics will be able to treat the residual expression as a lens between value environments and values, following [13,20]. Specifically, we have the *get* evaluation relation $\mu \vdash_{\mathrm{G}} E \Rightarrow v$, which computes the value v of E under environment μ as usual, and the *put* evaluation relation $\mu \vdash_{\mathrm{P}} v \Leftarrow E \dashv \mu'$, which computes an updated environment μ' for E from the updated view v and the original environment μ. In pseudo syntax, it can be understood as $put\ E\ \mu\ v = \mu'$, where μ represents the original source and μ' the new source.

It is worth mentioning that a complete separation of the stages is not possible due to the combination of **fix** and **case**, as an attempt to fully evaluate them in the first stage will result in divergence. Thus, we delay the unidirectional evaluation inside **case** to allow **fix**, and consequently the three evaluation relations (uni-directional, *get*, and *put*) are mutually dependent.

4.2 Three Evaluation Relations: Unidirectional, *get* and *put*

First, we formally define the set of residual expressions:

$$\begin{aligned} E ::= {} & \mathsf{True} \mid \mathsf{False} \mid [\,] \mid E_1 : E_2 \mid \lambda x.e \\ \mid {} & x \mid \underline{\mathsf{True}} \mid \underline{\mathsf{False}} \mid \underline{[\,]} \mid E_1 \mathbin{\underline{:}} E_2 \mid \underline{\mathbf{case}}\ E_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ E_i\ \underline{\mathbf{by}}\ E_i'\}_{i=1,2} \end{aligned}$$

They are treated as values in the unidirectional evaluation, and as expressions in the *get* and *put* evaluations. Notice that e or e_i appear under λ or **case**, meaning that their evaluations are delayed.

The set of *(first-order) values* is defined as below.

$$v ::= \mathsf{True} \mid \mathsf{False} \mid [\,] \mid v_1 : v_2$$

Accordingly, we define a *(first-order) value environment* μ as a finite mapping from variables to first-order values.

$$\frac{}{x \Downarrow_{\mathrm{U}} x} \qquad \frac{e_1 \Downarrow_{\mathrm{U}} \lambda x.e \quad e_2 \Downarrow_{\mathrm{U}} E_2 \quad e[E_2/x] \Downarrow_{\mathrm{U}} E}{e_1\ e_2 \Downarrow_{\mathrm{U}} E} \qquad \frac{}{\lambda x.e \Downarrow_{\mathrm{U}} \lambda x.e} \qquad \frac{e[\mathbf{fix}(\lambda f.e)/f] \Downarrow_{\mathrm{U}} E}{\mathbf{fix}(\lambda f.e) \Downarrow_{\mathrm{U}} E}$$

$$\frac{e_0 \Downarrow_{\mathrm{U}} E_0 \quad e_i' \Downarrow_{\mathrm{U}} E_i' \quad e_i'' \Downarrow_{\mathrm{U}} E_i'' \quad (i = 1, 2)}{\underline{\mathbf{case}}\ e_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ e_i'\ \underline{\mathbf{by}}\ e_i''\}_{i=1,2} \Downarrow_{\mathrm{U}} \underline{\mathbf{case}}\ E_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ E_i'\ \underline{\mathbf{by}}\ E_i''\}_{i=1,2}}$$

Fig. 5. Evaluation rules for unidirectional parts (excerpt)

Unidirectional Evaluation Relation. The rules for the unidirectional evaluation relation is rather standard, as excerpted in Fig. 5. The bidirectional constructs (i.e., bidirectional constructors and **case**) are frozen, i.e., behave just like ordinary constructors in this evaluation. Notice that we can evaluate an expression containing free variables; then the resulting residual expression may contain the free variables.

Bidirectional (*get* and *put*) Evaluation Relations. The *get* and *put* evaluation relations, $\mu \vdash_{\mathrm{G}} E \Rightarrow v$ and $\mu \vdash_{\mathrm{P}} v \Leftarrow E \dashv \mu'$, are defined so that they together form a lens.

Weakening of Environment. Before we lay out the semantics, it is worth explaining a subtlety in environment handling. In conventional evaluation semantics, a larger than necessary environment does no harm, as long as there is no name clashes. For example, whether the expression x is evaluated under the environment $\{x = 1\}$ or $\{x = 1, y = 2\}$ does not matter. However, the same is not true for bidirectional evaluation. Let us consider a residual expression $E = x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[\,]}$, and a value environment $\mu = \{x = 1, y = 2\}$ as the original source. We expect to have $\mu \vdash_{\mathrm{G}} E \Rightarrow 1 : 2 : [\,]$, which may be derived as:

$$\frac{\dfrac{}{\mu \vdash_{\mathrm{G}} x \Rightarrow 1} \quad \dfrac{\vdots}{\mu \vdash_{\mathrm{G}} y \mathbin{\underline{:}} \underline{[\,]} \Rightarrow 2 : [\,]}}{\mu \vdash_{\mathrm{G}} x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[\,]} \Rightarrow 1 : 2 : [\,]}$$

In the *put* direction, for an updated view say $3 : 4 : [\,]$, we expect to have $\mu \vdash_{\mathrm{P}} 3 : 4 : [\,] \Leftarrow E \dashv \{x = 3, y = 4\}$ with the corresponding derivation:

$$\frac{\dfrac{}{\mu \vdash_{\mathrm{P}} 3 \Leftarrow x \dashv ?_1} \quad \dfrac{\vdots}{\mu \vdash_{\mathrm{P}} 4 : [\,] \Leftarrow y \mathbin{\underline{:}} \underline{[\,]} \dashv ?_2}}{\mu \vdash_{\mathrm{P}} 3 : 4 : [\,] \Leftarrow x \mathbin{\underline{:}} y \mathbin{\underline{:}} \underline{[\,]} \dashv \{x = 3, y = 4\}}$$

What shall the environments $?_1$ and $?_2$ be? One way is to have $\mu \vdash_{\mathrm{P}} 3 \Leftarrow x \dashv \{x = 3, y = 2\}$, and $\mu \vdash_{\mathrm{P}} 4 : [\,] \Leftarrow y \mathbin{\underline{:}} \underline{[\,]} \dashv \{x = 1, y = 4\}$, where the variables do not appear free in the residual expression takes their values from the original source environment μ. However, the evaluation will get stuck here, as there is no reasonable way to produce the expected result $\{x = 3, y = 4\}$ from $?_1 = \{x = 3, y = 2\}$ and $?_2 = \{x = 1, y = 4\}$. In other words, the redundancy in environment is harmful as it may cause conflicts downstream.

Our solution to this problem, which follows from [21–23,29], is to allow *put* to return value environments containing only bindings that are relevant for the residual expressions under evaluation. For example, we have $\mu \vdash_{\mathrm{P}} 3 \Leftarrow x \dashv \{x = 3\}$, and $\mu \vdash_{\mathrm{P}} 4 : [\,] \Leftarrow y \mathbin{\underline{:}} [\,] \dashv \{y = 4\}$. Then, we can merge the two value environments $?_1 = \{x = 3\}$ and $?_2 = \{y = 4\}$ to obtain the expected result $\{x = 3, y = 4\}$. As a remark, this seemingly simple solution actually has a non-trivial effect on the reasoning of well-behavedness. We defer a detailed discussion on this to Sect. 4.3.

Now we are ready to define *get* and *put* evaluation rules for each bidirectional constructs. For variables, we just lookup or update environments. Recall that μ is a mapping (i.e., function) from variables to (first-order) values, while we use a record-like notation such as $\{x = v\}$.

$$\frac{}{\mu \vdash_{\mathrm{G}} x \Rightarrow \mu(x)} \qquad \frac{}{\mu \vdash_{\mathrm{P}} v \Leftarrow x \dashv \{x = v\}}$$

For constants $\underline{c}$ where $c = \mathsf{False}, \mathsf{True}, [\,]$, the evaluation rules are straightforward.

$$\frac{}{\mu \vdash_{\mathrm{G}} \underline{c} \Rightarrow c} \qquad \frac{}{\mu \vdash_{\mathrm{P}} c \Leftarrow \underline{c} \dashv \emptyset}$$

The above-mentioned behaviour of the bidirectional cons expression $E_1 \mathbin{\underline{:}} E_2$ is formally given as:

$$\frac{\mu \vdash_{\mathrm{G}} E_1 \Rightarrow v_1 \quad \mu \vdash_{\mathrm{G}} E_2 \Rightarrow v_2}{\mu \vdash_{\mathrm{G}} E_1 \mathbin{\underline{:}} E_2 \Rightarrow v_1 : v_2} \qquad \frac{\mu \vdash_{\mathrm{P}} v_1 \Leftarrow E_1 \dashv \mu_1' \quad \mu \vdash_{\mathrm{P}} v_2 \Leftarrow E_2 \dashv \mu_2'}{\mu \vdash_{\mathrm{P}} v_1 : v_2 \Leftarrow E_1 \mathbin{\underline{:}} E_2 \dashv \mu_1' \curlyvee \mu_2'}$$

(Note that the variable rules guarantee that only free variables in the residual expressions end up in the resulting environments.) Here, $\curlyvee$ is the merging operator defined as: $\mu \curlyvee \mu' = \mu \cup \mu'$ if there is no x such that $\mu(x) \neq \mu'(x)$. For example, $\{x = 3\} \curlyvee \{y = 4\} = \{x = 3, y = 4\}$, and $\{x = 3, y = 4\} \curlyvee \{y = 4\} = \{x = 3, y = 4\}$, but $\{x = 3, y = 2\} \curlyvee \{y = 4\}$ is undefined.

The most interesting rules are for **case**. In the *get* direction, it is not different from the ordinary **case** except that exit conditions are asserted, as shown in Fig. 6. We use the following predicate for pattern matching.

$$\mathit{match}(p_k, v_0, \mu_k) \;=\; (p_k\mu_k = v_0) \wedge (\mathsf{dom}(\mu_k) = \mathsf{fv}(p_k))$$

Here, we abuse the notation to write $p_k\mu_k$ for the value obtained from p_k by replacing the free variables x in p_k with $\mu_k(x)$. One might notice that we have the disjoint union $\mu \uplus \mu_i$ in Fig. 6 where μ_i holds the values of the variables in p_i, as we assume α-renaming of bound variables that is consistent in *get* and *put*. Recall that p_1 and p_2 are assumed not to overlap, and hence the evaluation is deterministic. Note that the reconciliation functions E_i'' are untouched by the rule.

The *put* evaluation rule of **case** shown in Fig. 6 is more involved. In addition to checking which branch should be chosen by using exit conditions, we need two rules to handle the cases with and without branch switching. Basically,

$$\frac{\mu \vdash_{\mathrm{G}} E_0 \Rightarrow v_0 \quad match(p_i, v_0, \mu_i) \quad e_i \Downarrow_{\mathrm{U}} E_i \quad \mu \uplus \mu_i \vdash_{\mathrm{G}} E_i \Rightarrow v \quad E_i' \; v \Downarrow_{\mathrm{U}} \mathsf{True}}{\mu \vdash_{\mathrm{G}} \underline{\mathbf{case}} \; E_0 \; \mathbf{of} \; \left\{p_i \to e_i \; \underline{\mathbf{with}} \; E_i' \; \underline{\mathbf{by}} \; E_i''\right\}_{i=1,2} \Rightarrow v}$$

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{G}} E_0 \Rightarrow v_0 \quad match(p_i, v_0, \mu_i) \quad E_i' \; v \Downarrow_{\mathrm{U}} \mathsf{True} \quad e_i \Downarrow_{\mathrm{U}} E_i \\ \mu \uplus \mu_i \vdash_{\mathrm{P}} v \Leftarrow E_i \dashv \mu' \uplus_{\mathsf{dom}(\mu),\mathsf{dom}(\mu_i)} \mu_i' \quad v_0' = p_i(\mu_i' \triangleleft \mu_i) \quad \mu \vdash_{\mathrm{P}} v_0' \Leftarrow E_0 \dashv \mu_0'\end{array}}{\mu \vdash_{\mathrm{P}} v \Leftarrow \underline{\mathbf{case}} \; E_0 \; \mathbf{of} \; \left\{p_i \to e_i \; \underline{\mathbf{with}} \; E_i' \; \underline{\mathbf{by}} \; E_i''\right\}_{i=1,2} \dashv \mu_0' \curlyvee \mu'}$$

$$\frac{\begin{array}{c}\mu \vdash_{\mathrm{G}} E_0 \Rightarrow v_0 \quad match(p_i, v_0, \mu_i) \quad E_i' \; v \Downarrow_{\mathrm{U}} \mathsf{False} \quad j = 3 - i \quad E_j' \; v \Downarrow_{\mathrm{U}} \mathsf{True} \quad e_j \Downarrow_{\mathrm{U}} E_j \\ E_j'' \; v_0 \; v \Downarrow_{\mathrm{U}} u_0 \quad match(p_j, u_0, \mu_j) \\ \mu \uplus \mu_j \vdash_{\mathrm{P}} v \Leftarrow E_j \dashv \mu' \uplus_{\mathsf{dom}(\mu),\mathsf{dom}(\mu_j)} \mu_j' \quad v_0' = p_j(\mu_j' \triangleleft \mu_j) \quad \mu \vdash_{\mathrm{P}} v_0' \Leftarrow E_0 \dashv \mu_0'\end{array}}{\mu \vdash_{\mathrm{P}} v \Leftarrow \underline{\mathbf{case}} \; E_0 \; \mathbf{of} \; \left\{p_i \to e_i \; \underline{\mathbf{with}} \; E_i' \; \underline{\mathbf{by}} \; E_i''\right\}_{i=1,2} \dashv \mu_0' \curlyvee \mu'}$$

Fig. 6. *get*- and *put*-Evaluation of **case**: we write $\mu \uplus_{X,Y} \mu'$ to ensure that $\mathsf{dom}(\mu) \subseteq X$ and $\mathsf{dom}(\mu') \subseteq Y$.

the branch to be taken in the backwards direction is decided first, by the *get*-evaluation of the case condition E_0 and the checking of the exit condition E_i' against the updated view v. After that, the body of the chosen branch e_i is firstly uni-directionally evaluated, and then its residual expression E_i is *put*-evaluated. The last step is *put*-evaluation of the case-condition E_0. When branch switching happens, there is the additional step of applying the reconciliation function E_j''.

Note the use of operator $\triangleleft$ in computing the updated case condition v_0'.

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

Recall that in the beginning of this subsection, we discussed our approach of avoiding conflicts by producing environments with only relevant variables. This means the μ_i' above contains only variables that appear free in E_i, which may or may not be all the variables in p_i. Since this is the point where these variables are introduced, we need to supplement μ_i' with μ_i from the original pattern matching so that p_i can be properly instantiated.

Construction of Lens. Let us write $\mathcal{L}_0[\![E]\!]$ for a lens between value environments and values, defined as:

$$\begin{array}{ll} get \; \mathcal{L}_0[\![E]\!] \; \mu = v & \text{if } \mu \vdash_{\mathrm{G}} E \Rightarrow v \\ put \; \mathcal{L}_0[\![E]\!] \; \mu \; v = \mu' & \text{if } \mu \vdash_{\mathrm{P}} v \Leftarrow E \dashv \mu' \end{array}$$

Then, we can define the lens $\mathcal{L}[\![e]\!]$ induced from e (a closed function expression), where $e \; x \Downarrow_{\mathrm{U}} E$ for some fresh variable x.

$$\begin{array}{l} get \; \mathcal{L}[\![e]\!] \; s = get \; \mathcal{L}_0[\![E]\!] \; \{x = s\} \\ put \; \mathcal{L}[\![e]\!] \; s \; v = (\mu' \triangleleft \{x = s\})(x) \quad \text{where } \mu' = put \; \mathcal{L}_0[\![E]\!] \; \{x = s\} \; v \end{array}$$

Actually, `:get` and `:put` in Sect. 2 are realised by $get \; \mathcal{L}[\![e]\!]$ and $put \; \mathcal{L}[\![e]\!]$.

4.3 Correctness

We establish the correctness of HOBiT Core: $\mathcal{L}[\![e]\!] \in \mathit{Lens}\ [\![\sigma]\!]\ [\![\tau]\!]$ is well-behaved for closed e of type $\mathbf{B}\sigma \to \mathbf{B}\tau$. Recall that $\mathit{Lens}\ S\ V$ is a set of lenses ℓ, where $\mathit{get}\ \ell \in S \to V$ and $\mathit{put}\ \ell \in S \to V \to S$. We only provide proof sketches in this subsection due to space limitation.

$\preceq$-well-behavedness. Recall that in the previous subsection, we allow environments to be weakened during *put*-evaluation. Since not all variables in a source may appear in the view, during some intermediate evaluation steps (for example within **case**-branches) the weakened environment may not be sufficient to fully construct a new source. Recall that, in $\mu \vdash_{\mathrm{P}} v \Leftarrow e \dashv \mu'$, $\mathsf{dom}(\mu')$ can be smaller than $\mathsf{dom}(\mu)$, a gap that is fixed at a later stage of evaluation by merging ($\curlyvee$) and defaulting ($\triangleleft$) with other environments. This technique reduces conflicts, but at the same time complicates the compositional reasoning of correctness. Specifically, due to the potentially missing information in the intermediate environments, well-behavedness may be temporally broken during evaluation. Instead, we use a variant of well-behavedness that is weakening aware, which will then be used to establish the standard well-behavedness for the final result.

Definition 1 ($\preceq$-well-behavedness). Let $(S, \preceq)$ and $(V, \preceq)$ be partially-ordered sets. A lens $\ell \in \mathit{Lens}\ S\ V$ is called *$\preceq$-well-behaved* if it satisfies

$$\mathit{get}\ \ell\ s = v \implies v \text{ is maximal} \wedge (\forall v'.\ v' \preceq v \implies \mathit{put}\ \ell\ s\ v' \preceq s) \qquad (\preceq\textbf{-Acceptability})$$

$$\mathit{put}\ \ell\ s\ v = s' \implies (\forall s''.\ s' \preceq s'' \implies v \preceq \mathit{get}\ \ell\ s'') \qquad (\preceq\textbf{-Consistency})$$

for any $s, s' \in S$ and $v \in V$, where s is maximal. □

We write $\mathit{Lens}^{\preceq\mathrm{wb}}\ S\ V$ for the set of lenses in $\mathit{Lens}\ S\ V$ that are $\preceq$-well-behaved. In this section, we only consider the case where S and V are value environments and first-order values, where value environments are ordered by weakening ($\mu \preceq \mu'$ if $\mu(x) = \mu'(x)$ for all $x \in \mathsf{dom}(\mu)$), and $(\preceq) = (=)$ for first-order values. In Sect. 5.2 we consider a slightly more general situation.

The $\preceq$-well-behavedness is a generalisation of the ordinary well-behavedness, as it coincides with the ordinary well-behavedness when $(\preceq) = (=)$.

Theorem 1. *For S and V with $(\preceq) = (=)$, a lens $\ell \in \mathit{Lens}\ S\ V$ is $\preceq$-well-behaved iff it is well-behaved.* □

Kripke Logical Relation. The key step to prove the correctness of HOBiT Core is to prove that $\mathcal{L}_0[\![E]\!]$ is always $\preceq$-well-behaved if E is an evaluation result of a well-typed expression e. The basic idea is to prove this by logical relation that expression e of type $\mathbf{B}\sigma$ under the context Δ is evaluated to E, assuming termination, such that $\mathcal{L}_0[\![E]\!]$ is a $\preceq$-well-behaved lens between $[\![\Delta]\!]$ and $[\![\sigma]\!]$.

Usually a logical relation is defined only by induction on the type. In our case, as we need to consider Δ in the interpretation of $\mathbf{B}\sigma$, the relation should be indexed by Δ too. However, naive indexing does not work due to substitutions.

For example, we could define a (unary) relation $\mathcal{E}_\Delta(\mathbf{B}\sigma)$ as a set of expressions that evaluate to "good" (i.e., $\preceq$-well-behaved) lenses between (the semantics of) Δ and σ, and $\mathcal{E}_\Delta(\mathbf{B}\sigma \to \mathbf{B}\tau)$ as a set of expressions that evaluate to "good" functions that map good lenses between Δ and σ to those between Δ and τ. This naive relation, however, does not respect substitution, which can substitute a value obtained from an expression typed under Δ to a variable typed under Δ' such that $\Delta \subseteq \Delta'$, where Δ and Δ' need not be the same. With the naive definition, good functions at Δ need not be good functions at Δ', as a good lens between Δ' and σ is not always a good lens between Δ and σ.

To remedy the situation, inspired by the denotation semantics in [24], we use Kripke logical relations [18] where worlds are Δs.

Definition 2. We define the set $\mathcal{E}_\Delta[\![A]\!]$ of expressions, the set $\mathcal{R}_\Delta[\![A]\!]$ of residual expressions, the set $[\![\sigma]\!]$ of values and the set $[\![\Delta]\!]$ of value environments as below.

$$
\begin{aligned}
\mathcal{E}_\Delta[\![A]\!] &= \{e \mid \forall E.\, e \Downarrow_{\mathrm{U}} E \text{ implies } E \in \mathcal{R}_\Delta[\![A]\!]\} \\
\mathcal{R}_\Delta[\![\mathit{Bool}]\!] &= \{\mathsf{True}, \mathsf{False}\} \\
\mathcal{R}_\Delta[\![[A]]\!] &= \mathit{List}\ \mathcal{R}_\Delta[\![A]\!] \\
\mathcal{R}_\Delta[\![\mathbf{B}\sigma]\!] &= \{E \mid \forall \Delta'.\, \Delta \subseteq \Delta' \text{ implies } \mathcal{L}_0[\![E]\!] \in \mathit{Lens}^{\preceq\mathrm{wb}}\ [\![\Delta']\!]\ [\![\sigma]\!]\} \\
\mathcal{R}_\Delta[\![A \to B]\!] &= \{F \mid \forall \Delta'.\, \Delta \subseteq \Delta' \text{ implies } (\forall E \in \mathcal{R}_{\Delta'}[\![A]\!].\, F\ E \in \mathcal{E}_{\Delta'}[\![B]\!])\} \\
[\![\mathit{Bool}]\!] &= \{\mathsf{True}, \mathsf{False}\} \\
[\![[\sigma]]\!] &= \mathit{List}\ [\![\sigma]\!] \\
[\![\Delta]\!] &= \{\mu \mid \mathsf{dom}(\mu) \subseteq \mathsf{dom}(\Delta) \text{ and } \forall x \in \mathsf{dom}(\mu).\mu(x) \in [\![\Delta(x)]\!]\}
\end{aligned}
$$

Here, for a set S, $\mathit{List}\ S$ is inductively defined as: $[\,] \in \mathit{List}\ S$, and $s : t \in \mathit{List}\ S$ for all $s \in S$ and $t \in \mathit{List}\ S$. □

The notable difference from ordinary logical relations is the definition of $\mathcal{R}_\Delta[\![A \to B]\!]$ where we consider an arbitrary Δ' such that $\Delta \subseteq \Delta'$. This is the key to state $\mathcal{R}_\Delta[\![A]\!] \subseteq \mathcal{R}_{\Delta'}[\![A]\!]$ if $\Delta \subseteq \Delta'$. Notice that $[\![\sigma]\!] = \mathcal{R}_\Delta[\![\sigma]\!]$ for any Δ.

We have the following lemmas.

Lemma 1. *If $\Delta \subseteq \Delta'$, $v \in \mathcal{R}_\Delta[\![A]\!]$ implies $v \in \mathcal{R}_{\Delta'}[\![A]\!]$.* □

Lemma 2. *$x \in \mathcal{R}_\Delta[\![\mathbf{B}\sigma]\!]$ for any Δ such that $\Delta(x) = \sigma$.* □

Lemma 3. *For any σ and Δ, $\underline{\mathsf{True}}, \underline{\mathsf{False}} \in \mathcal{R}_\Delta[\![\mathbf{B}\mathit{Bool}]\!]$ and $\underline{[\,]} \in \mathcal{R}_\Delta[\![\mathbf{B}[\sigma]]\!]$.* □

Lemma 4. *If $E_1 \in \mathcal{R}_\Delta[\![\mathbf{B}\sigma]\!]$ and $E_2 \in \mathcal{R}_\Delta[\![\mathbf{B}[\sigma]]\!]$, then $E_1 \mathbin{\underline{:}} E_2 \in \mathcal{R}_\Delta[\![\mathbf{B}[\sigma]]\!]$.* □

Lemma 5. *Let σ and τ be pure types and Δ a pure type environment. Suppose that $e_i \in \mathcal{E}_{\Delta \uplus \Delta_i}[\![\tau]\!]$ for $\Delta_i \vdash p_i : \sigma$ $(i = 1, 2)$, and that $E_0 \in \mathcal{R}_\Delta[\![\mathbf{B}\sigma]\!]$, $E_1', E_2' \in \mathcal{R}_\Delta[\![\tau \to \mathit{Bool}]\!]$ and $E_1'', E_2'' \in \mathcal{R}_\Delta[\![\sigma \to \tau \to \sigma]\!]$. Then, $\underline{\mathbf{case}}\ E_0\ \underline{\mathbf{of}}\ \{p_i \to e_i\ \underline{\mathbf{with}}\ E_i'\ \underline{\mathbf{by}}\ E_i''\}_{i=1,2} \in \mathcal{R}_\Delta[\![\mathbf{B}\tau]\!]$.*

Proof (Sketch). The proof itself is straightforward by case analysis. The key property is that *get* and *put* use the same branches in both proofs of $\preceq$-**Acceptability** and $\preceq$-**Consistency**. Slight care is required for unidirectional evaluations of e_1 and e_2, and applications of E_1', E_2', E_1'' and E_2''. However, the semantics is carefully designed so that in the proof of $\preceq$-**Acceptability**, unidirectional evaluations that happen in *put* have already happened in the evaluation of *get*, and a similar discussion applies to $\preceq$-**Consistency**. □

As a remark, recall that we assumed α-renaming of p_i so that the disjoint unions ($\uplus$) in Fig. 6 succeed. This renaming depends on the μs received in *get* and *put* evaluations, and can be realised by using de Bruijn levels.

Lemma 6 (Fundamental Lemma). *For $\Gamma; \Delta \vdash e : A$, for any Δ' with $\Delta \subseteq \Delta'$ and $E_x \in \mathcal{R}_{\Delta'}[\![\Gamma(x)]\!]$, we have $e[E_x/x]_x \in \mathcal{E}_{\Delta'}[\![A]\!]$.*

Proof (Sketch). We prove the lemma by induction on typing derivation. For bidirectional constructs, we just apply the above lemmas appropriately. The other parts are rather routine. □

Now we are ready to state the correctness of our construction of lenses.

Corollary 1. *If $\varepsilon; \varepsilon \vdash e : \mathbf{B}\sigma \to \mathbf{B}\tau$, then $e\ x \in \mathcal{E}_{\{x:\sigma\}}[\![\mathbf{B}\tau]\!]$.* □

Lemma 7. *If $e \in \mathcal{E}_{\{x:\sigma\}}[\![\mathbf{B}\tau]\!]$, $\mathcal{L}[\![e]\!]$ (if defined) is in $Lens^{\preceq \mathrm{wb}}\ [\![\sigma]\!]\ [\![\tau]\!]$ (and thus well-behaved by Theorem 1).* □

Theorem 2. *If $\varepsilon; \varepsilon \vdash e : \mathbf{B}\sigma \to \mathbf{B}\tau$, then $\mathcal{L}[\![e]\!] \in Lens\ [\![\sigma]\!]\ [\![\tau]\!]$ (if defined) is well-behaved.* □

5 Extensions

Before presenting a larger example, we discuss a few extensions of HOBiT Core which facilitate programming.

5.1 In-Language Lens Definition

In HOBiT programming, it is still sometimes useful to allow manually defined primitive lenses (i.e., lenses constructed from independently specified *get*/*put* functions), for backwards compatibility and also for programs with relatively simple computation logic but complicated backwards behaviours. This feature is supported by the construct **appLens** $e_1\ e_2\ e_3$ in HOBiT. For example, we can write $incB\ x = \underline{\mathbf{appLens}}\ \overline{(\lambda s.s+1)}\ (\lambda\_.\lambda v.v-1)\ x$ to define a bidirectional increment function $incB :: \mathbf{B}\mathit{Int} \to \mathbf{B}\mathit{Int}$. Note that for simplicity we require the

additional expression x (represented by e_3 in the general case) to convert between normal functions and lenses. The typing rule for $\underline{\textbf{appLens}}\ e_1\ e_2\ e_3$ is as below.

$$\frac{\Gamma;\Delta \vdash e_1 : \sigma \to \tau \quad \Gamma;\Delta \vdash e_2 : \sigma \to \tau \to \sigma \quad \Gamma;\Delta \vdash e_3 : \mathbf{B}\sigma}{\Gamma;\Delta \vdash \underline{\textbf{appLens}}\ e_1\ e_2\ e_3 : \mathbf{B}\tau}$$

Accordingly, we add the following unidirectional evaluation rule.

$$\frac{e_i \Downarrow_{\mathrm{U}} E_i \quad (i = 1, 2, 3)}{\underline{\textbf{appLens}}\ e_1\ e_2\ e_3 \Downarrow_{\mathrm{U}} \underline{\textbf{appLens}}\ E_1\ E_2\ E_3}$$

Also, we add the following *get*/*put* evaluation rules for $\underline{\textbf{appLens}}$.

$$\frac{\mu \vdash_{\mathrm{G}} E_3 \Rightarrow v \quad E_1\ v \Downarrow_{\mathrm{U}} u}{\mu \vdash_{\mathrm{G}} \underline{\textbf{appLens}}\ E_1\ E_2\ E_3 \Rightarrow u} \qquad \frac{\mu \vdash_{\mathrm{G}} E_3 \Rightarrow v \quad E_2\ v\ u' \Downarrow_{\mathrm{U}} v' \quad \mu \vdash_{\mathrm{P}} v' \Leftarrow E_3 \dashv \mu'}{\mu \vdash_{\mathrm{P}} u' \Leftarrow \underline{\textbf{appLens}}\ E_1\ E_2\ E_3 \dashv \mu'}$$

Notice that $\underline{\textbf{appLens}}\ e_1\ e_2\ e_3$ is "good" if e_3 is so, i.e., $\underline{\textbf{appLens}}\ e_1\ e_2\ e_3 \in \mathcal{E}_\Delta[\![\mathbf{B}\tau]\!]$ if $e_3 \in \mathcal{E}_\Delta[\![\mathbf{B}\sigma]\!]$, provided that the *get*/*put* pair (e_1, e_2) is well-behaved.

5.2 Lens Combinators as Language Constructs

In this paper, we have focused on the $\underline{\textbf{case}}$ construct, which is inspired by the *cond* combinator [7]. Although *cond* is certainly an important lens combinator, it is not the only one worth considering. Actually, we can obtain language constructs from a number of lens combinators including those that take care of alignment [2]. For the sake of demonstration, we outline the derivation of a simpler example $\mathit{comb} \in \mathit{Lens}\ [\![\sigma]\!]\ [\![\tau]\!] \to \mathit{Lens}\ [\![\sigma']\!]\ [\![\tau']\!]$. As the construction depends solely on types, we purposely leave the combinator abstract.

A naive way of lifting combinators can already be found in [21,23]. For example, for *comb*, we might prepare the construct $\underline{\textbf{comb}}_{\mathrm{bad}}$ with the following typing rule (where ε is the empty environment):

$$\frac{\varepsilon;\varepsilon \vdash e : \mathbf{B}\sigma \to \mathbf{B}\tau \quad \Gamma;\Delta \vdash e' : \mathbf{B}\tau'}{\Gamma;\Delta \vdash \underline{\textbf{comb}}_{\mathrm{bad}}\ e\ e' : \mathbf{B}\tau'}$$

Notice that in this version e is required to be closed so that we can turn the function directly into a lens by $\mathcal{L}[\![-]\!]$, and the evaluation of $\underline{\textbf{comb}}_{\mathrm{bad}}$ can then be based on standard lens composition: $\mathcal{L}_0[\![\underline{\textbf{comb}}_{\mathrm{bad}}\ E\ E']\!] = \mathit{comb}\ \mathcal{L}[\![E]\!] \mathbin{\hat{\circ}} \mathcal{L}_0[\![E']\!]$ (we omit the straightforward concrete evaluation rules), where E and E' is the unidirectional evaluation results of e and e' (notice that a residual expression is also an expression), and $\hat{\circ}$ is the lens composition combinator [7] defined by:

$$\begin{aligned}
&(\hat{\circ}) \in \mathit{Lens}\ B\ C \to \mathit{Lens}\ A\ B \to \mathit{Lens}\ A\ C \\
&\mathit{get}\ (\ell_2 \mathbin{\hat{\circ}} \ell_1)\ a \quad\ = \mathit{get}\ \ell_2\ (\mathit{get}\ \ell_1\ a) \\
&\mathit{put}\ (\ell_2 \mathbin{\hat{\circ}} \ell_1)\ a\ c' = \mathit{put}\ \ell_1\ a\ (\mathit{put}\ \ell_2\ (\mathit{get}\ \ell_1\ a)\ c')
\end{aligned}$$

The combinator preserves $\preceq$-well-behavedness, and thus $\underline{\textbf{comb}}_{\mathrm{bad}}$ guarantees correctness. However, as discussed extensively in the case of $\underline{\textbf{case}}$, this "closedness" requirements prevents flexible use of variables and creates a major obstacle in programming.

So instead of the plain *comb*, we shall assume a parameterised version $\mathit{pcomb} \in \mathit{Lens}\ (T \times [\![\sigma]\!])\ [\![\tau]\!] \to \mathit{Lens}\ (T \times [\![\sigma']\!])\ [\![\tau']\!]$ that allows each source to have an extra component T, which is expected to be kept track of by the combinator without modification. Here T is assumed to have a partial merging operator $(\curlyvee) \in T \to T \to T$ and a minimum element, and *pcomb* may use these facts in its definition. By using *pcomb*, we can give a corresponding language construct **comb** with a binder, typed as follows.

$$\frac{\Gamma; \Delta, x : \sigma \vdash e : \mathbf{B}\tau \quad \Gamma; \Delta \vdash e' : \mathbf{B}\sigma'}{\Gamma; \Delta \vdash \underline{\mathbf{comb}}\ (x.e)\ e' : \mathbf{B}\tau'}$$

We give its unidirectional evaluation rule as

$$\frac{e \Downarrow_{\mathrm{U}} E \quad e' \Downarrow_{\mathrm{U}} E'}{\underline{\mathbf{comb}}\ (x.e)\ e' \Downarrow_{\mathrm{U}} \underline{\mathbf{comb}}\ E\ E'}$$

We omit the *get*/*put* evaluation rules, which are straightforwardly obtained from the following equation.

$$\mathcal{L}_0[\![\underline{\mathbf{comb}}\ E\ E']\!] = \mathit{pcomb}\ (\mathit{unEnv}_x\ \mathcal{L}_0[\![E]\!]) \mathbin{\hat{\circ}} \langle \mathit{idL}, \mathcal{L}_0[\![E']\!] \rangle$$

where $\mathit{unEnv}_x \in \mathit{Lens}\ ([\![\Delta \uplus \{x : \sigma\}]\!])\ [\![\tau]\!] \to \mathit{Lens}\ ([\![\Delta]\!] \times [\![\sigma]\!])\ [\![\tau]\!]$ and $\langle -, - \rangle \in \mathit{Lens}\ [\![\Delta]\!]\ A \to \mathit{Lens}\ [\![\Delta]\!]\ B \to \mathit{Lens}\ [\![\Delta]\!]\ (A \times B)$ are lens combinators defined for any Δ as:

$$\begin{aligned}
&\mathit{get}\ (\mathit{unEnv}_x\ \ell)\ (\mu, v) &&= \mathit{get}\ \ell\ (\mu \uplus \{x = v\}) \\
&\mathit{put}\ (\mathit{unEnv}_x\ \ell)\ (\mu, v)\ u &&= (\mu', v') \\
&\quad \textbf{where}\ \mu' \uplus \{x = v'\} &&= (\mathit{put}\ \ell\ (\mu \uplus \{x = v\})\ v) \triangleleft \{x = v\} \\
&\mathit{get}\ \langle \ell_1, \ell_2 \rangle\ \mu &&= (\mathit{get}\ \ell_1\ \mu, \mathit{get}\ \ell_2\ \mu) \\
&\mathit{put}\ \langle \ell_1, \ell_2 \rangle\ \mu\ (a, b) &&= \mathit{put}\ \ell_1\ \mu\ a \curlyvee \mathit{put}\ \ell_2\ \mu\ b
\end{aligned}$$

Both combinators preserve $\preceq$-well-behavedness, where we assume the component-wise ordering on pairs. No "closedness" requirement is imposed on e in this version. From the construct, we can construct a higher-order function $\lambda f.\lambda z.\underline{\mathbf{comb}}\ (x.f\ x)\ z : (\mathbf{B}\sigma \to \mathbf{B}\tau) \to \mathbf{B}\sigma' \to \mathbf{B}\tau'$. That is, in HOBiT, lens combinators are just higher-order functions, as long as they permit the above-mentioned parameterisation. This observation means that we are able to systematically derive language constructs from lens combinators; as a matter of fact, the semantics of **case** is derived from a variant of the *cond* combinator [7].

Even better, the parametrised *pcomb* can be systematically constructed from the definition of *comb*. For *comb*, it is typical that *get* (*comb* ℓ) only uses *get* ℓ, and *put* (*comb* ℓ) uses *put* ℓ; that is, *comb* essentially consists of two functions of types $([\![\sigma]\!] \to [\![\tau]\!]) \to ([\![\sigma']\!] \to [\![\tau']\!])$ and $([\![\sigma]\!] \to [\![\tau]\!] \to [\![\sigma]\!]) \to ([\![\sigma']\!] \to [\![\tau']\!] \to [\![\sigma']\!])$. Then, we can obtain *pcomb* of the above type merely by "monad"ifying the two functions: using the reader monad $T \to -$ for the former and the composition of the reader and writer monads $T \to (-, T)$ backwards for the latter suffice to construct *pcomb*.

A remaining issue is to ensure that *pcomb* preserves $\preceq$-well-behavedness, which ensures $\underline{\textbf{comb}}\ (x.e)\ e' \in \mathcal{E}_\Delta[\![\mathbf{B}\tau']\!]$ under the assumptions $e \in \mathcal{E}_{\Delta \uplus \{x:\sigma\}}[\![\mathbf{B}\tau]\!]$ and $e' \in \mathcal{E}_\Delta[\![\mathbf{B}\sigma']\!]$. Currently, such a proof has to be done manually, even though *comb* preserves well-behavedness and *pcomb* is systematically constructed. Whether we can lift the correctness proof for *comb* to *pcomb* in a systematic way will be an interesting future exploration.

5.3 Guards

Guards used for branching are merely syntactic sugar in ordinary unidirectional languages such as Haskell. But interestingly, they actually increase the expressive power of HOBiT, by enabling inspection of updatable values without making the inspection functions bidirectional.

For example, Glück and Kawabe's reversible equivalence check [10] can be implemented in HOBiT as follows.

$$
\begin{array}{l}
\mathit{eqCheck} :: \mathbf{B}\sigma \to \mathbf{B}\sigma \to \mathbf{B}(\mathit{Either}\ (\sigma,\sigma)\ \sigma) \\
\mathit{eqCheck}\ x\ y = \underline{\textbf{case}}\ \underline{(}x, y\underline{)}\ \underline{\textbf{of}} \\
\quad (x', y') \mid x' == y' \ \to \underline{\mathsf{Right}}\ x' \qquad \underline{\textbf{with}}\ \mathit{isRight}\ \underline{\textbf{by}}\ (\lambda\_.\lambda(\mathsf{Right}\ x).(x,x)) \\
\quad (x', y') \mid \mathit{otherwise} \to \underline{\mathsf{Left}}\ \underline{(}x', y'\underline{)}\ \underline{\textbf{with}}\ \mathit{isLeft} \quad \underline{\textbf{by}}\ (\lambda\_.\lambda(\mathsf{Left}\ (x,y)).(x,y))
\end{array}
$$

Here, $\underline{(}-,-\underline{)}$ is the bidirectional version of the pair constructor. The exit condition *isRight* checks whether a value is headed by the constructor Right, and *isLeft* by Left. Notice that the backwards transformation of *eqCheck* fails when the updated view is $\mathsf{Left}\ (v, v)$ for some v.

5.4 Syntax Sugar for Reconciliation Functions

In the general form, reconciliation functions take in two arguments for the computation of the new source. But as we have seen, very often the arguments are not used in the definition and therefore redundant. This observation motivates the following syntax sugar.

$$p \to e\ \underline{\textbf{with}}\ e'\ \underline{\textbf{default}}\ \{x_1 = e_1''; \ldots; x_n = e_n''\}$$

Here, $x_1, \ldots, x_n$ are the free variables in p. This syntax sugar is translated as:

$$p \to e\ \underline{\textbf{with}}\ e'\ \underline{\textbf{by}}\ \lambda\_.\lambda\_.p[e_1''/x_1, \ldots, e_n''/x_n]$$

Furthermore, it is also possible to automatically derive some default values from their types. This idea can be effectively implemented if we extend HOBiT with type classes.

5.5 Inference of Exit Conditions

It is possible to infer exit conditions from their surrounding contexts; an idea that has been studied in the literature of invertible programming [11,20], and may benefit from range analysis.

Our prototype implementation adopts a very simple inference that constructs an exit condition $\lambda x.\mathbf{case}\ x\ \mathbf{of}\ \{p_e \to \mathsf{True}; \_ \to \mathsf{False}\}$ for each branch, where p_e is the skeleton of the branch body e, constructed by replacing bidirectional constructors with the unidirectional counterparts, and non-constructor expressions with $\_$. For example, from $a : \mathit{appendB}\ x'\ y$, we obtain the pattern $\_ : \_$. This embarrassingly simple inference has proven to be handy for developing larger HOBiT programs as we will see in Sect. 6.

6 An Involved Example: Desugaring

In this section, we demonstrate the programmability of HOBiT using the example of bidirectional desugaring [26]. Desugaring is a standard process for most programming languages, and making it bidirectional allows information in desugared form to be propagated back to the surface programs. It is argued convincingly in [26] that such bidirectional propagation (coined *resugaring*) is effective in mapping reduction sequences of desugared programs into those of the surface programs.

Let us consider a small programming language that consists of **let**, **if**, Boolean constants, and predefined operators.

$$\mathbf{data}\ E = \mathsf{ELet}\ E\ E \mid \mathsf{EVar}\ \mathit{Int} \mid \mathsf{EIf}\ E\ E\ E \mid \mathsf{ETrue} \mid \mathsf{EFalse} \mid \mathsf{EOp}\ \mathit{Name}\ [E]$$
$$\mathbf{type}\ \mathit{Name} = \mathit{String}$$

Variables are represented as de Bruijn indices.

Some operators in this language are syntactic sugar. For example, we may want to desugar

$$\mathsf{EOp}\ \texttt{"not"}\ [e] \qquad \text{as} \qquad \mathsf{EIf}\ e\ \mathsf{EFalse}\ \mathsf{ETrue}.$$

Also, e_1 `||` e_2 can be transformed to $\mathbf{let}\ x = e_1\ \mathbf{in\ if}\ x\ \mathbf{then}\ x\ \mathbf{else}\ e_2$, which in our mini-language is the following.

$$\mathsf{EOp}\ \texttt{"or"}\ [e_1, e_2] \qquad \text{as} \qquad \mathsf{ELet}\ e_1\ (\mathsf{EIf}\ (\mathsf{EVar}\ 0)\ (\mathsf{EVar}\ 0)\ (\mathit{shift}\ 0\ e_2))$$

Here, *shift* n is the standard shifting operator for de Brujin indexed-term that increments the variables that have indices greater than n (these variables are "free" in the given expression). We will program a bidirectional version of the above desugaring process in Figs. 7 and 8, with the particular goal of keeping the result of a backward execution as close as possible to the original sugared form (so that it is not merely a "decompilation" in the sense that the original source has to be consulted).

```
composB :: (BE → BE) → BE → BE
composB f x = case x of
  EIf e1 e2 e3 → EIf (f e1) (f e2) (f e3)   by recE
  ELet e1 e2   → ELet (f e1) (f e2)         by recE
  EVar n       → EVar n                     by recE
  ETrue        → ETrue
  EFalse       → EFalse
  EOp n es     → EOp n (mapB ETrue f es)    by recE
mapB :: a → (Ba → Bb) → B[a] → B[b]
mapB def z = case z of
  []     → []
  a : x → f a : mapB def x default {a = def; x = []}

recE :: E → E → E
recE e (EIf _ _ _) = EIf ETrue e e
recE e (ELet _ _)  = ELet e (shift 0 e)
recE e (EOp n _)   = toOp n e
recE e e'          = e'
toOp :: Name → E → E
toOp n e =
  let k = fromJust (lookup n arities)
  in EOp (replicate k e)
```

Fig. 7. *composB*: a useful building block

```
shiftB :: Int → BE → BE
shiftB n e = case e of
  ELet e1 e2       → ELet (shiftB n e1) (shiftB (n + 1) e2)   default {e1 = ETrue; e2 = EFalse}
  EVar m | m < n → EVar m                  with varLT n        default m = 0
  EVar m | m ≥ n → EVar (incB m)           with varGT n        default m = n + 1
  e'               → composB (shiftB n) e' with nonLetVar by recE

desugarB :: BE → BE
desugarB e = case e of
  EOp "or" [e1, e2] → ELet (desugarB e1) (EIf (EVar 0) (EVar 0) (desugarB (shiftB 0 e2)))
                                           by (λs.λ_.toOp "or" s)
  EOp "not" [e]     → EIf e EFalse ETrue   by (λs.λ_.toOp "not" s)
  e'                → composB desugarB e' by recE

varLT n (EVar m) = m < n          nonLetVar (ELet _ _) = False
varLT n _        = False          nonLetVar (EVar _)   = False
varGT n (EVar m) = m > n          nonLetVar e          = True
varGT n _        = False
```

Fig. 8. *desugarB*: bidirectional desugring

We start with an auxiliary function *compos* [4] in Fig. 7, which is a useful building block for defining shifting and desugaring. We have omitted the straightforward exit conditions; they will be inferred as explained in Sect. 5.5. The function *mapB* is the bidirectional map. The reconciliation function *recE* tries to preserves as much source structure as possible by reusing the original source e. Here, $arities :: [(Name, Int)]$ maps operator names to their arities (i.e. $arities = [(\texttt{"or"}, 2), (\texttt{"not"}, 1)]$). The function *shift* is the standard uni-directional shifting function. We omit its definition as it is similar to the bidirectional version in Fig. 8. Note that **default** is syntactic sugar for reconciliation function introduced in Sect. 5.4. Here, *incB* is the bidirectional increment function defined in Sect. 5.1. Thanks to *composB*, we only need to define the interesting parts in the definitions of *shiftB* and *desugarB*. The reconciliation

functions *recE* and *toOp* try to keep as much source information as possible, which enables the behaviour that the backwards execution produces "`not`" and "`or`" in the sugared form only if the original expression has the sugar.

Consider a sugared expression EOp `"or"` [EOp `"not"` [ETrue], EOp `"not"` [EFalse]] as a source *source*.

```
HOBiT> :get desugarB source
ELet (EIf ETrue EFalse ETrue) (EIf (EVar 0) (EVar 0) (EIf EFalse EFalse ETrue)
{- let x = (if True then False else True)
   in if x then x else (if False then False else True) -}
```

The following updated views may be obtained by reductions from the view.

```
{- view1 ≡ let x = False in if x then x else (if False then False else True) -}
view1 = ELet EFalse (EIf (EVar 0) (EVar 0) (EIf EFalse EFalse ETrue)
{- view2 ≡ if False then False else (if False then False else True) -}
view2 = EIf EFalse EFalse (EIf EFalse EFalse ETrue)
{- view3 ≡ if False then False else True -}
view3 = EIf EFalse EFalse ETrue
```

The following are the corresponding backward transformation results.

```
HOBiT> :put desugarB source view1
EOp "or" [EFalse, EOp "not" [EFalse]]
HOBiT> :put desugarB source view2
EIf EFalse EFalse (EOp "not" [EFalse]
HOBiT> :put desugarB source view3
EOp "not" [False]
```

As the AST structure of the view is changed, all of the three cases require branch-switching in the backwards executions; our program handles it with ease. For *view*$_2$, the top-level expression EIf EFalse EFalse ... does not have a corresponding sugared form. Our program keeps the top level unchanged, and proceeds to the subexpression with correct resugaring, a behaviour enabled by the appropriate use of reconciliation function (the first line of *recE* for this particular case) in *composB*.

If we were to present the above results as the evaluation steps in the surface language, one may argue that the second result above does not correspond to a valid evaluation step in the surface language. In [26], AST nodes introduced in desugaring are marked with the information of the original sugared syntax, and resugaring results containing the marked nodes will be skipped, as they do not correspond to any reduction step in the surface language. The marking also makes the backwards behaviour more predictable and stable for drastic changes on the view, as the desugaring becomes injective with this change. This technique is orthogonal to our exploration here, and may be combined with our approach.

7 Related Work

Controlling Backwards Behaviour. In addition to $put \in S \to V \to S$, many lens languages [3] supply a $create \in V \to S$ (which is in essence a right-inverse of *get*) to be used when the original source data is unavailable. This happens when new data is inserted in the view, which does not have any corresponding source for *put* to execute, or when branch-switching happens but with no reconciliation function available. Being a right-inverse, *create* does not fail (assuming it terminates), but since it is not guided by the original source, the results are more arbitrary. We do not include *create* in HOBiT, as it complicates the system without offering obvious benefits. Our branch-switching facilities are perfectly capable of handling missing source data via reconciliation functions.

Using exit conditions in branching constructs for backwards evaluation can be found in a number of related fields: bidirectional transformation [7], reversible computation [34] and program inversion [11,20]. Our design of **case** is inspired by the *cond* combinator in the lens framework [7] and the `if`-statement in Janus [34]. A similar combinator is *Case* in BiGUL [16], where a branch has a function performing a similar role as an exit condition, but taking the original source in addition. This difference makes *Case* more expressive than *cond*; for example, *Case* can implement matching lenses [2]. Our design of **case** follows *cond* for its relative simplicity, but the same underlying technique can be applied to *Case* as mentioned in Sect. 5.2. In the context of *bidirectionalization* [19,29,30] there is the idea of "Plug-ins" [31] that are similar to reconciliation functions in the sense that source values can be adapted to direct backwards execution.

Applicative Lenses. The applicative lens framework [21,23] provides a way to use λ-abstraction and function application as in normal functional programming to compose lenses. Note that this use of "applicative" refers to the classical applicative (functional) programming style, and is not directly related to `Applicative` functor in Haskell. In this sense, it shares a similar goal to us. But crucially, applicative lens lacks HOBiT's ability to allow λ-bound variables to be used freely, and as a result suffers from the same limitation of lens languages. There are also a couple of technical differences between applicative lens and our work: applicative lens is based on Yoneda embedding while ours is based on separating Γ and Δ and having three semantics (Sect. 4); and applicative lens is implemented as an embedded DSL, while HOBiT is given as a standalone language. Embedded implementation of HOBiT is possible, but a type-correct embedding would expose the handling of environment Δ to programmers, which is undesirable.

Lenses and Their Extensions. As mentioned in Sect. 1, the most common way to construct lenses is by using combinators [3,7,8], in which lenses are treated as opaque objects and composed by using lens combinators. Our goal in this paper is to enhance the programmability of lens programming, while keeping its expressive power as possible. In HOBiT, primitive lenses can be represented as functions on **B**-typed values (Sect. 5.1), and lens combinators satisfying certain conditions can be represented as language construct with binders (Sect. 5.2), which is at least enough to express the original lenses in [7].

Among extensions of the lens language [2,3,7–9,16,17,27,32], there exists a few that extend the classical lens model [7], namely quotient lenses [8], symmetric lenses [14], and edit-based lenses [15]. A natural question to ask is whether our development, which is based on the classical lenses, can be extended to them. The answer depends on treatment of value environments μ in *get* and *put*. In our semantics, we assume a non-linear system as we can use the same variable in μ any number of times. This requires us to extend the classical lens to allow merging ($\curlyvee$) and defaulting ($\triangleleft$) operations in *put* with $\preceq$-well-behavedness, but makes the syntax and type system of HOBiT simple, and HOBiT free from the design issues of linear programming languages [25]. Such extension of lenses would be applicable to some kinds of lens models, including quotient lenses and symmetric lenses, but its applicability is not clear in general. Also, we want to mention that allowing duplications in bidirectional transformation is still open, as it essentially entails multiple views and the synchronization among them.

8 Conclusion

We have designed HOBiT, a higher-order bidirectional programming language in which lenses are represented as functions and lens combinators are represented as language constructs with binders. The main advantage of HOBiT is that users can program in a style similar to conventional functional programming, while still enjoying the benefits of lenses (i.e., the expressive power and well-behavedness guarantee). This has allowed us to program realistic examples with relative ease.

HOBiT for the first time introduces a truly "functional" way of constructing bidirectional programs, which opens up a new area of future explorations. Particularly, we have just started to look at programming techniques in HOBiT. Moreover, given the resemblance of HOBiT code to that in conventional languages, the application of existing programming tools becomes plausible.

Acknowledgements. We thank Shin-ya Katsumata, Makoto Hamana and Kazuyuki Asada for their helpful comments on the category theory and denotational semantics, from which our formal discussions originate. The work was partially supported by JSPS KAKENHI Grant Numbers 24700020, 15K15966, and 15H02681.

References

1. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Trans. Database Syst. **6**(4), 557–575 (1981). https://doi.org/10.1145/319628.319634
2. Barbosa, D.M.J., Cretin, J., Foster, N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: Hudak, P., Weirich, S. (eds.) ICFP, pp. 193–204. ACM (2010). https://doi.org/10.1145/1863543.1863572
3. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: Necula, G.C., Wadler, P. (eds.) POPL, pp. 407–419. ACM (2008). https://doi.org/10.1145/1328438.1328487
4. Bringert, B., Ranta, A.: A pattern for almost compositional functions. J. Funct. Program. **18**(5–6), 567–598 (2008). https://doi.org/10.1017/S0956796808006898

5. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM **48**(3), 555–604 (2001). https://doi.org/10.1145/382780.382785
6. Fegaras, L.: Propagating updates through XML views using lineage tracing. In: Li, F., Moro, M.M., Ghandeharizadeh, S., Haritsa, J.R., Weikum, G., Carey, M.J., Casati, F., Chang, E.Y., Manolescu, I., Mehrotra, S., Dayal, U., Tsotras, V.J. (eds.) ICDE, pp. 309–320. IEEE (2010). https://doi.org/10.1109/ICDE.2010.5447896
7. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Trans. Program. Lang. Syst. **29**(3) (2007). https://doi.org/10.1145/1232420.1232424
8. Foster, J.N., Pilkiewicz, A., Pierce, B.C.: Quotient lenses. In: Hook, J., Thiemann, P. (eds.) ICFP, pp. 383–396. ACM (2008). https://doi.org/10.1145/1411204.1411257
9. Foster, N., Matsuda, K., Voigtländer, J.: Three complementary approaches to bidirectional programming. In: Gibbons, J. (ed.) Generic and Indexed Programming. LNCS, vol. 7470, pp. 1–46. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32202-0_1
10. Glück, R., Kawabe, M.: A program inverter for a functional language with equality and constructors. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 246–264. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-40018-9_17
11. Glück, R., Kawabe, M.: Revisiting an automatic program inverter for lisp. SIGPLAN Not. **40**(5), 8–17 (2005). https://doi.org/10.1145/1071221.1071222
12. Hegner, S.J.: Foundations of canonical update support for closed database views. In: Abiteboul, S., Kanellakis, P.C. (eds.) ICDT 1990. LNCS, vol. 470, pp. 422–436. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-53507-1_93
13. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing graph transformations. In: Hudak, P., Weirich, S. (eds.) ICFP, pp. 205–216. ACM (2010). https://doi.org/10.1145/1863543.1863573
14. Hofmann, M., Pierce, B.C., Wagner, D.: Symmetric lenses. In: Ball, T., Sagiv, M. (eds.) POPL, pp. 371–384. ACM (2011). https://doi.org/10.1145/1926385.1926428
15. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: Field, J., Hicks, M. (eds.) POPL, pp. 495–508. ACM (2012). https://doi.org/10.1145/2103656.2103715
16. Hu, Z., Ko, H.S.: Principles and practice of bidirectional programming in BiGUL. Oxford Summer School on Bidirectional Transformations (2017). https://bitbucket.org/prl_tokyo/bigul/raw/master/SSBX16/tutorial.pdf. Accessed 18 Oct 2017
17. Hu, Z., Mu, S.-C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. In: Heintze, N., Sestoft, P. (eds.) PEPM, pp. 178–189. ACM (2004). https://doi.org/10.1145/1014007.1014025
18. Jung, A., Tiuryn, J.: A new characterization of lambda definability. In: Bezem, M., Groote, J.F. (eds.) TLCA 1993. LNCS, vol. 664, pp. 245–257. Springer, Heidelberg (1993). https://doi.org/10.1007/BFb0037110
19. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: Hinze, R., Ramsey, N. (eds.) ICFP, pp. 47–58. ACM (2007). https://doi.org/10.1145/1291151.1291162
20. Matsuda, K., Mu, S.-C., Hu, Z., Takeichi, M.: A grammar-based approach to invertible programs. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 448–467. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11957-6_24
21. Matsuda, K., Wang, M.: Applicative bidirectional programming: mixing lenses and semantic bidirectionalization. J. Funct. Program. Accepted 14 Feb 2018

22. Matsuda, K., Wang, M.: "Bidirectionalization for free" for monomorphic transformations. Sci. Comput. Program. **111**(1), 79–109 (2014). https://doi.org/10.1016/j.scico.2014.07.008
23. Matsuda, K., Wang, M.: Applicative bidirectional programming with lenses. In: Fisher, K., Reppy, J.H. (eds.) ICFP, pp. 62–74. ACM (2015). https://doi.org/10.1145/2784731.2784750
24. Moggi, E.: Functor categories and two-level languages. In: Nivat, M. (ed.) FoSSaCS 1998. LNCS, vol. 1378, pp. 211–225. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0053552
25. Morris, J.G.: The best of both worlds: linear functional programming without compromise. In: Garrigue, J., Keller, G., Sumii, E. (eds.) ICFP, pp. 448–461. ACM (2016). https://doi.org/10.1145/2951913.2951925
26. Pombrio, J., Krishnamurthi, S.: Resugaring: lifting evaluation sequences through syntactic sugar. In: O'Boyle, M.F.P., Pingali, K. (eds.) PLDI, pp. 361–371. ACM (2014). https://doi.org/10.1145/2594291.2594319
27. Rajkumar, R., Foster, N., Lindley, S., Cheney, J.: Lenses for web data. ECEASST **57** (2013). https://doi.org/10.14279/tuj.eceasst.57.879
28. Stevens, P.: A landscape of bidirectional model transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88643-3_10
29. Voigtländer, J.: Bidirectionalization for free! (pearl). In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 165–176. ACM (2009). https://doi.org/10.1145/1480881.1480904
30. Voigtländer, J., Hu, Z., Matsuda, K., Wang, M.: Combining syntactic and semantic bidirectionalization. In: Hudak, P., Weirich, S. (eds.) ICFP, pp. 181–192. ACM (2010). https://doi.org/10.1145/1863543.1863571
31. Voigtländer, J., Hu, Z., Matsuda, K., Wang, M.: Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. J. Funct. Program. **23**(5), 515–551 (2013). https://doi.org/10.1017/S0956796813000130
32. Wang, M., Gibbons, J., Matsuda, K., Hu, Z.: Refactoring pattern matching. Sci. Comput. Program. **78**(11), 2216–2242 (2013). https://doi.org/10.1016/j.scico.2012.07.014
33. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) ASE, pp. 164–173. ACM (2007). https://doi.org/10.1145/1321631.1321657
34. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Ramírez, A., Bilardi, G., Gschwind, M. (eds.) CF, pp. 43–54. ACM (2008). https://doi.org/10.1145/1366230.1366239
35. Yu, Y., Lin, Y., Hu, Z., Hidaka, S., Kato, H., Montrieux, L.: Maintaining invariant traceability through bidirectional transformations. In: Glinz, M., Murphy, G.C., Pezzè, M. (eds.) ICSE, pp. 540–550. IEEE (2012). https://doi.org/10.1109/ICSE.2012.6227162