Automated Synthesis of Functional Programs with Auxiliary Functions

Shingo Eguchi, Naoki Kobayashi, and Takeshi Tsukada

The University of Tokyo

Abstract. Polikarpova et al. have recently proposed a method for synthesizing functional programs from specifications expressed as refinement types, and implemented a program synthesis tool Synquid. Although Synquid can generate non-trivial programs on various data structures such as lists and binary search trees, it cannot automatically generate programs that require auxiliary functions, unless users provide the specifications of auxiliary functions. We propose an extension of Synquid to enable automatic synthesis of programs with auxiliary functions. The idea is to prepare a template of the target function containing unknown auxiliary functions, infer the types of auxiliary functions, and then use Synquid to synthesize the auxiliary functions. We have implemented a program synthesizer based on our method, and confirmed through experiments that our method can synthesize several programs with auxiliary functions, which Synquid is unable to automatically synthesize.

1 Introduction

The goal of program synthesis [2–4, 6, 7, 9, 10] is to automatically generate programs from certain program specifications. The program specifications can be examples (a finite set of input/output pairs) [2, 3], validator code [10], or refinement types [9]. In the present paper, we are interested in the approach of synthesizing programs from refinement types [9], because refinement types can express detailed specifications of programs, and synthesized programs are guaranteed to be correct by construction (in that they indeed satisfy the specification given in the form of refinement types).

Polikarpova et al. [9] have formalized a method for synthesizing a program from a given refinement type, and implemented a program synthesis tool called Synquid. It can automatically generate a number of interesting programs such as those manipulating lists and trees. Synquid, however, suffers from the limitation that it cannot automatically synthesize programs that require auxiliary functions (unless the types of auxiliary functions are given as hints).

In the present paper, we propose an extension of Synquid to enable automatic synthesis of programs with auxiliary functions. Given a refinement type specification of a function, our method proceeds as follows.

Step 1: Prepare a template of the target function with unknown auxiliary functions. The template is chosen based on the simple type of the target function.

$$sort :: \; l : \texttt{List}\ Int \\ \rightarrow \{\texttt{List}\ Int\ \langle \lambda x.\lambda y.x \leq y \rangle \mid \texttt{len}\ \nu = \texttt{len}\ l\ \wedge\ \texttt{elems}\ \nu = \texttt{elems}\ l\}$$

Fig. 1. The type of a sorting function

$$\begin{aligned} sort = \lambda l.\ & \texttt{match}\ l\ \texttt{with} \\ & \mid \texttt{Nil} \mapsto \Box_1 \\ & \mid \texttt{Cons}\ x\ xs \mapsto \Box_2\ x\ (sort\ xs\ ) \end{aligned}$$

Fig. 2. A template for list-sorting function

For example, if the function takes a list as an argument, a template that recurses over the list is typically selected.

Step 2: Infer the types of auxiliary functions from the template.

Step 3: Synthesize the auxiliary functions by passing the inferred types to SYNQUID. (If this fails, go back to Step 1 and choose another template.)

We sketch our method through an example of the synthesis of a list-sorting function. Following SYNQUID [9], a specification of the target function can be given as the refinement type shown in Figure 1. Here, "$\texttt{List}\ Int\ \langle \lambda x.\lambda y.x \leq y \rangle$" is the type of a sorted list of integers, where the part $\lambda x.\lambda y.x \leq y$ means that $(\lambda x.\lambda y.x \leq y) v_1\, v_2$ holds for any two elements v_1 and v_2 such that v_1 occurs before v_2 in the list. Thus, the type specification in Figure 1 means that the target function *sort* should take a list of integers as input, and returns a sorted list that is of the same length and has the same set of elements as the input list.

In Step 1, we generate a template of the target function. Since the argument of the function is a list, a default choice is the "fold template" shown in Figure 2. The template contains the holes $\Box_1$ and $\Box_2$ for unknown auxiliary functions. Thus, the goal has been reduced to the problem of finding appropriate auxiliary functions to fill the holes.

In Step 2, we infer the types of auxiliary functions, so that the whole function has the type in Figure 2. This is the main step of our method and consists of a few substeps. First, using a variation of the type inference algorithm of SYNQUID, we obtain type judgments for the auxiliary functions. For example, for $\Box_2$, we infer:

$$\begin{aligned} & l : \texttt{List}\ Int, x : Int, xs : \{\texttt{List}\ Int \mid \texttt{len}\ \nu = \texttt{len}\ l - 1 \wedge \texttt{elems}\ \nu + [x] = \texttt{elems}\ l\} \\ & \vdash \Box_2 :: x' : \{Int \mid \nu = x\} \\ & \qquad \rightarrow l' : \{\texttt{List}\ Int\ \langle \lambda x.\lambda y.x \leq y \rangle \mid \texttt{len}\ \nu = \texttt{len}\ xs\ \wedge\ \texttt{elems}\ \nu = \texttt{elems}\ xs\} \\ & \qquad \rightarrow \{\texttt{List}\ Int\ \langle \lambda x.\lambda y.x \leq y \rangle \mid \texttt{len}\ \nu = \texttt{len}\ l\ \wedge\ \texttt{elems}\ \nu = \texttt{elems}\ l\}. \end{aligned}$$

Here, for example, the type of the second argument of $\Box_2$ comes from the type of the target function *sort*. Since we wish to infer a *closed* function for $\Box_2$

$$\Box_1 :: \{\texttt{List}\ Int\ \langle \lambda x \lambda y.x \leq y\rangle \mid \texttt{len}\ \nu = 0\ \wedge\ \texttt{elems}\ \nu = \emptyset\}$$
$$\Box_2 :: x : Int \to l : \texttt{List}\ Int$$
$$\to \{\texttt{List}\ Int\ \langle \lambda x \lambda y.x \leq y\rangle \mid \texttt{len}\ \nu = \texttt{len}\ l + 1\ \wedge\ \texttt{elems}\ \nu = \texttt{elems}\ l + [x]\}$$

Fig. 3. The type of the auxiliary function

$$\begin{aligned}
&g = \lambda x.\lambda l.\texttt{match}\ l\ \texttt{with}\\
&\qquad \mid \texttt{Nil} \mapsto \texttt{Cons}x\texttt{Nil}\\
&\qquad \mid \texttt{Cons}\ y\ ys \mapsto\ \texttt{if}\ x\ \leq y\ \texttt{thenCons}\ x\ (\texttt{Cons}\ y\ ys)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \texttt{else}\ \texttt{Cons}\ y\ (g\ x\ ys)\\
&sort = \lambda l.\texttt{match}\ l\ \texttt{with}\\
&\qquad \mid \texttt{Nil}\ \mapsto \texttt{Nil}\\
&\qquad \mid \texttt{Cons}\ x\ xs \mapsto g\ x\ (sort\ xs)
\end{aligned}$$

Fig. 4. A synthesized list-sorting function

(that does not contain l, x, xs), we then convert the above judgment to a closed type using quantifiers. For example, the result type becomes:

$$\begin{aligned}
&\{\texttt{List}\ Int\ \langle \lambda x.\lambda y.x \leq y\rangle \mid\\
&\quad \forall l, x, xs.(\texttt{len}\ xs = \texttt{len}\ l - 1 \wedge \texttt{elems}\ xs + [x] = \texttt{elems}\ l\\
&\qquad \wedge \texttt{len}\ l' = \texttt{len}\ xs\ \wedge\ \texttt{elems}\ l' = \texttt{elems}\ xs)\\
&\qquad \Rightarrow \texttt{len}\ \nu = \texttt{len}\ l\ \wedge\ \texttt{elems}\ \nu = \texttt{elems}\ l\}.
\end{aligned}$$

Here, the lefthand side of the implication comes from the constraints in the type environment and the type of the second argument. We then eliminate quantifiers (in a sound but incomplete manner), and obtain the types shown in Figure 3.

Finally, in Step 3, we just pass the inferred types of auxiliary functions to Synquid. By filling the holes of the template with the auxiliary functions synthesized by Synquid, we get a complete list-sorting function as shown in Figure 4.

We have implemented a prototype program synthesis tool, which uses Synquid as a backend, based on the proposed method. We have tested it for several examples, and confirmed that our method is able to synthesize programs with auxiliary functions, which Synquid alone fails to synthesize automatically.

The rest of the paper is structured as follows. Section 2 defines the target language. Section 3 describes the proposed method. Section 4 reports an implementation and experimental results. Section 5 discusses related work and Section 6 concludes the paper.

$$
\begin{array}{l}
t \text{ (program terms)} ::= e \;\mid b \;\mid f \\
e \text{ (E-terms)} \;\; ::= x \mid e\ e \mid e\ f \\
b \text{ (branching)} \;\; ::= \texttt{if}\ e\ \texttt{then}\ t\ \texttt{else}\ t \mid (\texttt{match}\ e\ \texttt{with}\ \texttt{C}_1 \widetilde{x}_1 \;\mapsto t_1 \mid \cdots \mid \texttt{C}_k \widetilde{x}_k \;\mapsto t_k) \\
f \text{ (functions)} \;\; ::= \lambda x.t \mid \texttt{fix}\ x.t
\end{array}
$$

Fig. 5. Syntax of programs

2 Target Language

This section defines the target language of program synthesis. Since the language is essentially the same as the one used in SYNQUID [9], we explain it only briefly. For the sake of simplicity, we omit polymorphic types in the formalization below, although they are supported by the implementation reported in Section 4.

Figure 5 shows the syntax of program terms. Following [9], we classify terms into E-terms, branching, and function terms; this is for the convenience of formalizing the synthesis algorithm. Apart from it, the syntax is that of a standard functional language. In the figure, x and $\texttt{C}$ range over the sets of variables and data constructors respectively. Data constructors are also treated as variables (so that $\texttt{C}$ is also an E-term). The match expression first evaluates e, and if the value is of the form $\texttt{C}_i\,\widetilde{v}$, evaluates $[\widetilde{v}/\widetilde{x}_i]t_i$; here we write $\widetilde{\cdot}$ for a sequence. The function term $\texttt{fix}\ x.t$ denotes the recursive function defined by $x = t$.

The syntax of types is given in Figure 6. A type is either a refinement type $\{B \mid \psi\}$ or a function type $x : T_1 \to T_2$. The type $\{B \mid \psi\}$ describes the set of elements ν of ground type B that satisfies ψ; here, ψ is a formula that may contain a special variable ν, which refers to the element. For example, $\{\mathit{Int} \mid \nu > 0\}$ represents the type of an integer ν such that $\nu > 0$. For a technical convenience, we assume that ψ always contains ν as a free variable, by considering $\psi \land (\nu = \nu)$ instead of ψ if necessarily. The function type $x{:}T_1 \to T_2$ is dependent, in that x may occur in T_2 when T_1 is a refinement type. A ground type B is either a base type (*Bool* or *Int*), or a data type $D\,T_1 \cdots T_n$, where D denotes a type constructor. For the sake of simplicity, we consider only covariant type constructors, i.e., $D\,T_1 \cdots T_n$ is a subtype of $D\,T'_1 \cdots T'_n$ if T_i is a subtype of T'_i for every $i \in \{1, \ldots, n\}$. The type $\texttt{List}\ \mathit{Int}\ \langle \lambda x.\lambda y.x \leq y\rangle$ of sorted lists in Section 1 is expressed as $(\texttt{List}\langle \lambda x.\lambda y.x \leq y\rangle)\mathit{Int}$, where $\texttt{List}\langle \lambda x.\lambda y.x \leq y\rangle$ is the D-part. The list constructor *Cons* is given a type of the form:

$$
\begin{array}{l}
z : \{B \mid \psi'\} \to w : (\texttt{List}\langle \lambda x.\lambda y.\psi\rangle)\{B \mid \psi' \land [z/x, \nu/y]\psi\} \\
\quad \to \{(\texttt{List}\langle \lambda x.\lambda y.\psi\rangle)\{B \mid \psi'\} \mid \texttt{len}\ \nu = \texttt{len}\ w + 1 \land \texttt{elems}\ \nu = \texttt{elems}\ w + [z]\}
\end{array}
$$

for each ground type B and formulas ψ, ψ'. Here, $\texttt{len}$ and $\texttt{elems}$ are uninterpreted function symbols. In a contextual type $\texttt{let}\ C\ \texttt{in}\ T$, the context C binds some variables in T and impose constraints on them; for example, $\texttt{let}\ x : \{\mathit{Int} \mid \nu > 0\}\ \texttt{in}\ \{\mathit{Int} \mid \nu = 2x\}$ denotes the type of positive even integers.

T (types) $::= \{B \mid \psi\} \mid x : T_1 \to T_2$
B (ground types) $::= \mathit{Bool} \mid \mathit{Int} \mid D\, T_1 \cdots T_n$
C (contexts) $::= \cdot \mid x : T; C$
$\hat{T}$ (contextual types) $::= \texttt{let}\ C\ \texttt{in}\ T$

Fig. 6. Syntax of types

A *type environment* Γ is a sequence consisting of bindings of variables to types and formulas (called *path conditions*), subject to certain well-formedness conditions. We write $\Gamma \vdash T$ to mean that T is well formed under Γ; see Appendix A for the well-formedness conditions on types and type environments. Figure 7 shows the typing rules. The typing rules are fairly standard ones for a refinement type system, except that, in rule T-App, contextual types are used to avoid substituting program terms for variables in types; this treatment of contextual types follows the formalization of Synquid [9].

In the figure, $\mathsf{FV}(\psi)$ represents the set of free variables occurring in ψ. In rule T-Match, $\widetilde{x_i} : \widetilde{T_i} \to T$ represents $x_{i,1} : T_{i,1} \to \cdots x_{i,k_i} : T_{i,k_i} \to T$.

We write $[\![\Gamma]\!]_{vars}$ for the formula obtained by extracting constraints on the variables *vars* from Γ. It is defined by:

$$\begin{aligned}
[\![\Gamma; \psi]\!]_{vars} &= \psi \wedge [\![\Gamma]\!]_{vars \cup \mathsf{FV}(\psi)} \\
[\![\Gamma; x : \{B \mid \psi\}]\!]_{vars} &= \begin{cases} [x/\nu]\psi \wedge [\![\Gamma]\!]_{vars \cup \mathsf{FV}(\psi)} & \text{if } x \in vars \\ [\![\Gamma]\!]_{vars} & \text{otherwise} \end{cases} \\
[\![\Gamma; x : T_1 \to T_2]\!]_{vars} &= [\![\Gamma]\!]_{vars} \\
[\![\cdot]\!]_{vars} &= \top.
\end{aligned}$$

The goal of our program synthesis is, given a type environment Γ (that represents the types of constants and already synthesized functions) and a type T, to find a program term t such that $\Gamma \vdash t :: T$.

3 Our Method

This section describes our method for synthesizing programs with auxiliary functions. As mentioned in Section 1, the method consists of the following three steps:

Step 1: Generate a program template with unknown auxiliary functions.
Step 2: Infer the types of the unknown auxiliary functions.
Step 3: Synthesize auxiliary functions of the required types by using Synquid.

3.1 Step 1: Generating templates

In this step, program templates are generated based on the (simple) type of an argument of the target function. Figure 8 shows the syntax of templates. It is an extension of the language syntax described in Section 2 with unknown auxiliary functions $\Box_i$. We require that for each i, $\Box_i$ occurs only once in a template.

Subtyping $\boxed{\Gamma \vdash T <: T'}$

$$\frac{\Gamma \vdash B <: B' \qquad \mathsf{valid}(\llbracket \Gamma \rrbracket_{\mathsf{FV}(\psi \to \psi')} \land \psi \to \psi')}{\Gamma \vdash \{B \mid \psi\} <: \{B' \mid \psi'\}} \quad (<:\text{-G})$$

$$\frac{\Gamma \vdash T_1 <: T_1' \qquad (\Gamma; y : T_1) \vdash [y/x]T_2' <: T_2}{\Gamma \vdash x : T_1' \to T_2' <: y : T_1 \to T_2} \quad (<:\text{-}\textsc{Fun})$$

$$\frac{}{\Gamma \vdash \mathit{Int} <: \mathit{Int}} \quad (<:\text{-}\textsc{Int}) \qquad\qquad \frac{}{\Gamma \vdash \mathit{Bool} <: \mathit{Bool}} \quad (<:\text{-}\textsc{Bool})$$

$$\frac{\Gamma \vdash T_i <: T_i' \text{ for each } i \in \{1, \ldots, n\}}{\Gamma \vdash D\ T_1\ \cdots\ T_n <: D\ T_1'\ \cdots\ T_n'} \quad (<:\text{-DT})$$

Typing with contextual types $\boxed{\Gamma \vdash e :: \hat{T}}$

$$\frac{\Gamma(x) = \{B \mid \psi\}}{\Gamma \vdash x :: \mathtt{let}\ \cdot\ \mathtt{in}\ \{B \mid \nu = x\}} \quad (\text{T-}\textsc{VarG})$$

$$\frac{\Gamma(x) = T \qquad \Gamma \vdash T}{\Gamma \vdash x :: \mathtt{let}\ \cdot\ \mathtt{in}\ T} \quad (\text{T-}\textsc{Var})$$

$$\frac{\begin{array}{c}\Gamma \vdash e :: \mathtt{let}\ C_1\ \mathtt{in}\ x : T_x \to T \\ \Gamma; C_1 \vdash t :: \mathtt{let}\ C_2\ \mathtt{in}\ T_x' \\ \Gamma; C_1; C_2 \vdash T_x' <: T_x\end{array}}{\Gamma \vdash e\ t :: \mathtt{let}\ C_1; C_2; x : T_x'\ \mathtt{in}\ T} \quad (\text{T-}\textsc{App})$$

Context-free typing $\boxed{\Gamma \vdash t :: T}$

$$\frac{\Gamma \vdash e :: \mathtt{let}\ C\ \mathtt{in}\ T' \qquad \Gamma; C \vdash T' <: T}{\Gamma \vdash e :: T} \quad (\text{T-}\textsc{Sub})$$

$$\frac{\Gamma \vdash x : T_x \to T \qquad \Gamma; x : T_x \vdash t :: T}{\Gamma \vdash \lambda x.t :: x : T_x \to T} \quad (\text{T-}\textsc{Abs})$$

$$\frac{\begin{array}{c}\Gamma \vdash e :: \mathtt{let}\ C\ \mathtt{in}\ \{\mathtt{Bool} \mid \psi\} \qquad \Gamma \vdash T \\ \Gamma; C; [\top/\nu]\psi \vdash t_1 :: T \qquad \Gamma; C; [\bot/\nu]\psi \vdash t_2 :: T\end{array}}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 :: T} \quad (\text{T-}\textsc{If})$$

$$\frac{\begin{array}{c}\Gamma \vdash e :: \mathtt{let}\ C\ \mathtt{in}\ \{D\ T_1'\ \cdots\ T_n' \mid \psi\} \qquad \Gamma \vdash T \\ \Gamma(\mathtt{C}_i) = \widetilde{x}_i : \widetilde{T}_i\ \to \{D\ T_1'\ \cdots\ T_n' \mid \psi_i'\} \qquad \Gamma_i = \widetilde{x}_i : \widetilde{T}_i; [z/\nu]\psi_i' \\ \Gamma; C; z : \{D\ T_1'\ \cdots\ T_n' \mid \psi\}; \Gamma_i \vdash t_i :: T \text{ (for each } i\text{)}\end{array}}{\Gamma \vdash \mathtt{match}\ e\ \mathtt{with}\ \mathtt{C}_1\widetilde{x}_1\ \mapsto t_1 \mid \cdots \mid \mathtt{C}_k\widetilde{x}_k\ \mapsto t_k :: T} \quad (\text{T-}\textsc{Match})$$

$$\frac{\Gamma; x : T \vdash t :: T}{\Gamma \vdash \mathtt{fix}\ x.t :: T} \quad (\text{T-}\textsc{Fix})$$

Fig. 7. Typing rules

$$
\begin{array}{rl}
t_\square \text{ (program terms with holes)} & ::= e \mid b \mid f \mid e_\square \mid b_\square \mid f_\square \\
e_\square \text{ (E-terms with a hole)} & ::= \square_i \mid e_\square\ e \mid e_\square\ f \\
b_\square \text{ (branching with holes)} & ::= \texttt{if}\ e\ \texttt{then}\ t_\square\ \texttt{else}\ t_\square \\
 & \mid\ (\texttt{match}\ e\ \texttt{with}\ \texttt{C}_1\widetilde{x}_1 \mapsto t^1_\square \mid \cdots \mid \texttt{C}_k\widetilde{x}_k \mapsto t^k_\square) \\
f_\square \text{ (functions with holes)} & ::= \lambda x.t_\square \mid \texttt{fix}\ x.t_\square
\end{array}
$$

Fig. 8. The syntax of templates

We generate multiple candidates of templates automatically, and proceed to Steps 2 and 3 for each candidate. If the synthesis fails, we backtrack and try another candidate.

In the current implementation (reported in Section 4), we prepare the following templates.

– Fold-style (or, catamorphism) templates: These are templates of functions that recurse over an argument of algebraic data type. For example, the followings are templates for unary functions on lists (shown on the lefthand side) and those on binary trees (shown on the righthand side).

$$
\begin{array}{ll}
f = \lambda l.\ \texttt{match}\ l\ \texttt{with} & f = \lambda t.\ \texttt{match}\ t\ \texttt{with} \\
\quad \texttt{Nil} \mapsto \square_1 & \quad \texttt{Empty} \mapsto \square_1 \\
\quad \mid \texttt{Cons}\ x\ xs \mapsto \square_2\ x\ (f\ xs) & \quad \mid \texttt{Node}\ v\ l\ r \mapsto \square_2\ x\ (f\ l)\ (f\ r)
\end{array}
$$

– Divide-conquer-style templates: These are templates for functions on lists (or other set-like data structures). The following is a template for a function that takes a list as the first argument.

$$
\begin{array}{l}
f = \lambda l.\ \texttt{match}\ l\ \texttt{with} \\
\quad \texttt{Nil} \mapsto \square_1 \\
\quad \mid \texttt{Cons}\ x\ \texttt{Nil} \mapsto \square_2\ x \\
\quad \mid \texttt{Cons}\ x\ xs \mapsto (\texttt{match}\ (\mathit{split}\ l)\ \texttt{with}\ \texttt{Pair}\ l_1\ l_2 \mapsto \square_3\ (f\ l_1)\ (f\ l_2))
\end{array}
$$

The function f takes a list l as an input; if the length of l is more than 1, it splits l into two lists l_1 and l_2, recursively calls itself for l_1 and l_2, and combines the result with the unknown auxiliary function $\square_3$. A typical example that fits this template is the merge sort function, where $\square_3$ is the merge function.

Note that the rest of our method (Steps 2 and 3) does not depend on the choice of templates; thus other templates can be freely added.

3.2 Step 2: Inferring the types of auxiliary functions

This section describes a procedure to infer the types of auxiliary functions from the template generated in Step 1. This procedure is the core part of our method, which consists of the following three substeps.

Step 2.1: Extract type constraints on each auxiliary function.
Step 2.2: From the type constraints, construct closed types of auxiliary functions that may contain quantifiers in refinement formulas.
Step 2.3: Eliminate quantifiers from the types of auxiliary functions.

Step 2.1: Extraction of type constraints Given a type T of a program to synthesize and a program template $t_\square$ with n holes, this step derives a set $\{\Gamma_1 \vdash \square_1 :: T_1, \ldots, \Gamma_n \vdash \square_n :: T_n\}$ of constraints for each hole $\square_i$. The constraints mean that, if each hole $\square_i$ is filled by a closed term of type stronger than T_i, then the resulting program has type T.

The procedure is shown in Fig. 9, obtained based on the typing rules in Section 2. It is similar to the type checking algorithm used in Synquid [9]; the main difference from the corresponding type inference algorithm of Synquid is that, when a template of the form $\square_i\, e_1\, \ldots\, e_n$ is encountered (the case for $e_\square$ in the procedure `step2.1`, processed by the subprocedure `extractConst`), we first perform type inference for the arguments $e_1, \ldots, e_n$, and then construct the type for $\square_i$. To see this, observe that the template $\square_i\, e_1\, \ldots\, e_n$ matches the first pattern $e_\square :: T$ of the match expression in $\mathsf{step2.1}$, and the subprocedure $\mathsf{extractConst}$ is called. In $\mathsf{extractConst}$, $\square_i\, e_1\, \ldots\, e_n$ (with $n > 0$) matches the second pattern $e'_\square\, e$ (where $e'_\square$ and e are bound to $\square_i\, e_1\, \ldots\, e_{n-1}$ and e_n respectively), and the type T_n of e_n is first inferred. Subsequently, the procedure $\mathsf{extractConst}$ is recursively called and the types $T_{n-1}, \ldots, T_1$ of $e_{n-1}, \ldots, e_1$ (along with contexts $C_{n-1}, \ldots, C_1$) are inferred in this order, and then $y_1 : T_1 \to \cdots \to y_n : T_n \to T$ (along with a context) is obtained the type of $\square_i$. In contrast, for an application $e_1 e_2$, Synquid first performs type inference for the function part e_1, and then propagates the resulting type information to the argument e_2.

Example 1. Given the type T of a sorting function in Figure 1 and the template $t_\square$ in Figure 2, $\mathsf{step2.1}(\Gamma \vdash t_\square :: T)$ (where Γ contains types for constants such as `Nil`) returns the following constraint for the auxiliary function $\square_2$ (we omit types for constants).

$$
\begin{array}{l}
l : \texttt{List}\ \mathit{Int}\ ; x : \mathit{Int};\ xs : \texttt{List}\ \langle \lambda x.\lambda y.x \leq y \rangle\ \{\mathit{Int} \mid x \leq \nu\}\ ; \\
z : \{\texttt{List}\ \mathit{Int} \mid \nu = l\}; \texttt{len}\ xs + 1 = \texttt{len}\ z \land \texttt{elems}\ xs + [x] = \texttt{elems}\ z \\
\vdash \\
\square_i :: y : \{\mathit{Int} \mid \nu = x\} \\
\qquad \to ys : \{\texttt{List}\langle \lambda x \lambda y.x\ \leq y\rangle\ \{\mathit{Int} \mid x\ \leq \nu\} \\
\qquad\qquad\qquad \mid \texttt{len}\ \nu = \texttt{len}\ xs \land \texttt{elems}\ \nu = \texttt{elems}\ xs\} \\
\qquad \to \{\texttt{List}\langle \lambda x \lambda y.x\ \leq y\rangle\ \mathit{Int} \mid \texttt{len}\ \nu = \texttt{len}\ l \land \texttt{elems}\ \nu = \texttt{elems}\ l\}.
\end{array}
$$

□

$\mathsf{step2.1}(\Gamma \vdash\ t_\square :: T)\ =$

$\texttt{match}\ (t_\square :: T)\ \texttt{with}$

$\quad \mid e_\square :: T \quad \Rightarrow \quad \mathsf{extractConst}(\Gamma,\ e_\square,\ T\ , \emptyset)$

$\quad \mid e :: T \qquad \texttt{when} \quad \Gamma \vdash e :: T \qquad \Rightarrow\ \emptyset$

$\quad \mid \texttt{fix}\ x.t_\square\ :: T \quad \Rightarrow \quad \mathsf{step2.1}\ ((\Gamma; x : T) \vdash t_\square : T)$

$\quad \mid \lambda y.t_\square :: (x : T_x \to T') \quad \Rightarrow \quad \mathsf{step2.1}((\Gamma; y : T_x) \vdash t_\square :: [y/x]T') \qquad (1)$

$\quad \mid \texttt{if}\ e_1\ \texttt{then}\ t'_\square\ \texttt{else}\ t''_\square\ : T$

$\quad\quad \texttt{when} \quad \Gamma \vdash e_1 :: \texttt{let}\ C\ \texttt{in}\ \{\texttt{Bool} \mid \psi\} \Rightarrow$

$\quad\quad\quad \mathsf{step2.1}((\Gamma; C; [\texttt{true}/\nu]\psi) \vdash t'_\square : T) \cup \mathsf{step2.1}((\Gamma; C; [\texttt{false}/\nu]\psi) \vdash t''_\square : T)$

$\quad \mid (\texttt{match}\ e\ \texttt{with}\ \texttt{C}_1 \widetilde{x}_1\ \mapsto t^{(1)}_\square \mid \cdots \mid \texttt{C}_k \widetilde{x}_k\ \mapsto t^{(k)}_\square) : T$

$\quad\quad \texttt{when} \quad \Gamma \vdash e :: \texttt{let}\ C\ \texttt{in}\ \{D\ \widetilde{T} \mid \psi\}$

$\quad\quad\quad \Gamma(\texttt{C}_i) = \widetilde{x}_i : \widetilde{T}_i\ \to \{D\ \widetilde{T} \mid \psi'_i\} \Rightarrow$

$\quad\quad\quad \bigcup_i \mathsf{step2.1}((\Gamma; C; z : \{D\ \widetilde{T} \mid \psi\} \vdash t^{(i)}_\square : T)$ (where z is fresh)

$\quad \mid \_\ \Rightarrow\ \mathit{fail}$

$\mathsf{extractConst}(\Gamma,\ e_\square,\ T\ , C)\ =$

$\quad \texttt{match}\ e_\square\ \texttt{with}$

$\quad\quad \mid \square_i \Rightarrow \{\Gamma; C \vdash \square_i :: T\}$

$\quad\quad \mid e'_\square\ e \Rightarrow$

$\quad\quad\quad$ infer C' and T' such that

$\quad\quad\quad\quad \Gamma \vdash e :: \texttt{let}\ C'\ \texttt{in}\ T'$

$\quad\quad\quad\quad$ where all variables bounded in C' occur only in T'

$\quad\quad\quad \mathsf{extractConst}(\Gamma,\ e'_\square,\ y : T' \to T\ , C; C')$ (where y is fresh)

$\quad\quad \mid e'_\square\ f \Rightarrow$

$\quad\quad\quad$ infer T' such that

$\quad\quad\quad \Gamma \vdash f :: T'$

$\quad\quad\quad \mathsf{extractConst}(\Gamma,\ e'_\square,\ y : T' \to T\ , C)$ (where y is fresh)

Fig. 9. The algorithm for Step 2.1

The theorem below states the soundness of the procedure. Intuitively, it claims that a target program of type T can indeed be obtained from a given template $t_\Box$, by filling the holes $\Box_1, \ldots, \Box_n$ with terms $t_1, \ldots, t_n$ of the types inferred by the procedure `step2.1`.

Theorem 1. *Let Γ be a well-formed environment, $t_\Box$ a program template and T a type well-formed under Γ. Suppose that* `step2.1`$(\Gamma \vdash t_\Box :: T)$ *returns*

$$\{\Delta_1 \vdash \Box_1 :: U_1, \ldots, \Delta_n \vdash \Box_n :: U_n\}.$$

If $\emptyset \vdash S_i$ and $\Delta_i \vdash S_i <: U_i$ for each $i \in \{1, \ldots, n\}$, then

$$\Gamma; \Box_1 : S_1, \ldots, \Box_n : S_n \vdash t_\Box :: T.$$

Step 2.2: Construction of closed types We have obtained a constraint $\Gamma_i \vdash \Box_i :: T_i$ for each hole $\Box_i$, and now it suffices to find an auxiliary function (i.e. a closed term) of type T_i for each i. We shall use Synquid [9] to synthesize a desired function but the type T_i itself cannot be an input of Synquid since it is not closed in general. The goal of Step 2.2 is, thus, to calculate a closed type S_i such that $\Gamma \vdash S_i <: T_i$, using universal and existential quantifiers.

In order to solve the problem above by induction on T_i, we generalize the problem as follows: Given a well-formed type $\Gamma \vdash T$ and a set *var* of variables,

(a) find a type S such that $\Gamma \vdash S <: T$ and $\mathsf{FV}(S) \subseteq var$, and
(b) find a type S such that $\Gamma \vdash T <: S$ and $\mathsf{FV}(S) \subseteq var$.

Let us first consider the simplest but most important case, where T is a scalar type $\{B \mid \psi\}$ with $B = Bool$ or Int. Suppose that ψ has free variables $\{\nu\} \cup var \cup \{y_1, \ldots, y_n\}$, where y_i $(1 \le i \le n)$ comes from the environment Γ and $y_i \notin var$. Let $var = \{x_1, \ldots, x_k\}$ and $\boldsymbol{x}$ be the sequence of variables $x_1, \ldots, x_k$. The goal is to find a formula $\psi_0(\nu, \boldsymbol{x})$ with free variable $\{\nu, x_1, \ldots, x_k\}$ such that

$$\Gamma \vdash \{B \mid \psi_0(\nu, \boldsymbol{x})\} <: \{B \mid \psi(\nu, \boldsymbol{x}, \boldsymbol{y})\}.$$

By the subtyping rule, this subtyping judgment holds if and only if

$$[\![\Gamma]\!]_{\boldsymbol{x},\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \wedge \psi_0(\nu, \boldsymbol{x}) \Rightarrow \psi(\nu, \boldsymbol{x}, \boldsymbol{y})$$

is valid. The weakest formula $\psi_0(\nu, \boldsymbol{x})$ that satisfies the above condition can be given by using the universal quantifier, namely,

$$\psi_0(\nu, \boldsymbol{x}) \quad := \quad \forall \boldsymbol{y}\boldsymbol{z}.\, \big([\![\Gamma]\!]_{\boldsymbol{x},\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \Rightarrow \psi(\nu, \boldsymbol{x}, \boldsymbol{y})\big).$$

The dual problem can be solved in a similar way: the formula $\psi'_0(\nu)$ defined by

$$\psi'_0(\nu, \boldsymbol{x}) \quad := \quad \exists \boldsymbol{y}\boldsymbol{z}.\, \big([\![\Gamma]\!]_{\boldsymbol{x},\boldsymbol{y}}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}) \wedge \psi(\nu, \boldsymbol{x}, \boldsymbol{y})\big)$$

satisfies the subtyping judgment $\Gamma \vdash \{B \mid \psi(\nu, \boldsymbol{x}, \boldsymbol{y})\} <: \{B \mid \psi'_0(\nu, \boldsymbol{x})\}$.

The case $T = \{D\,U_1\,\ldots\,U_\ell \mid \psi\}$ is similar to the above case, except that we should replace each U_i with a closed type S_i. We recursively call the procedure to construct such a S_i.

When $T = (x : T_1 \to T_2)$, we simply invoke the procedures recursively. Every solution S must be of the form $S = (x : S_1 \to S_2)$, and the requirements are $\Gamma \vdash T_1 <: S_1$ (with $\mathsf{FV}(S_1) \subseteq \mathit{var}$) and $\Gamma; x : T_1 \vdash S_2 <: T_2$ (with $\mathsf{FV}(S_2) \subseteq \mathit{var} \cup \{x\}$). These subproblems can be solved by recursively calling the procedure.

Figure 10 gives a formal definition of the procedures; $\mathsf{necessType}(\Gamma \vdash T,\ \mathit{var})$ solves the problem (a) and $\mathsf{suffType}(\Gamma \vdash T,\ \mathit{var})$ does (b).

Example 2. We continue discussing the example of the list sorting function. So far, the following constraint for the hole $\square_2$ is derived. (Γ is same as the environment shown in Example 1)

$$\begin{aligned}\Gamma \vdash\ \square_2 :: y : \{\mathit{Int} \mid \nu = x\} \\ &\to ys : \{\mathtt{List}\langle \lambda x \lambda y.x\ \le y\rangle\ \{\mathit{Int} \mid x\ \le \nu\} \\ &\qquad\qquad \mid \mathtt{len}\ \nu = \mathtt{len}\ xs \land \mathtt{elems}\ \nu = \mathtt{elems}\ xs\} \\ &\to \{\mathtt{List}\langle \lambda x \lambda y.x\ \le y\rangle\ \mathit{Int} \mid \mathtt{len}\ \nu = \mathtt{len}\ l \land \mathtt{elems}\ \nu = \mathtt{elems}\ l\}\end{aligned}$$

In this step, we construct a closed type from the above constraint. The result is shown in Figure 11.

The type returned by the procedure indeed satisfies the requirement.

Theorem 2. *Let $\Gamma \vdash T$ be a well-formed type and var be a set of variables.*

- *If $S = \mathsf{necessType}(\Gamma \vdash T,\ \mathit{var})$, then $\Gamma \vdash S <: T$ and $\mathsf{FV}(S) \subseteq \mathit{var}$.*
- *If $S = \mathsf{suffType}(\Gamma \vdash T,\ \mathit{var})$, then $\Gamma \vdash T <: S$ and $\mathsf{FV}(S) \subseteq \mathit{var}$.*

Hence, if $S = \mathsf{necessType}(\Gamma \vdash T,\ \emptyset)$, then $\Gamma \vdash S <: T$ and S is closed.

Step 2.3: Elimination of quantifiers By Step 2.2, closed types of auxiliary functions have been obtained, but these types cannot be passed to Synquid yet because Synquid can handle only types with quantifier-free refinement formulas. Therefore, in Step 2.3, we eliminate quantifiers from the types derived by Step 2.2. Depending on the underlying logic, there may not exist a sound and complete quantifier elimination procedure. For example, in our running example, we use a combination of uninterpreted function symbols, linear integer arithmetic, and sets, for which a complete procedure does not exist. We thus apply a sound but incomplete procedure, so that, given the type T obtained by Step 2.2, produces a subtype T' of T that does not contain quantifiers.

An important observation in designing a sound procedure is that, by the definition of the procedure for Step 2.2, existential quantifiers may occur in the form $\exists \widetilde{x}.(\psi_1 \land \cdots \land \psi_k)$ only in *negative* positions of types, and universal quantifiers may occur in the form $\forall \widetilde{x}.(\psi_1 \land \cdots \land \psi_k \Rightarrow \psi)$ only in *positive* positions. Here, as usual, we say that ψ occurs positively in $\{B \mid \psi\}$, and that ψ occurs positively (resp. negatively) in $x{:}T_1 \to T_2$ if ψ occurs positively (resp. negatively)

$\mathsf{step2.2}(\mathit{\Gamma} \vdash T) \; = \; \mathsf{necessType}(\mathit{\Gamma} \vdash T, \; \emptyset)$

$\mathsf{necessType}(\mathit{\Gamma} \vdash T, \; \mathit{var}) \; =$
$\quad \texttt{match } T \texttt{ with}$
$\qquad \mid \{B \mid \psi\} \quad \Rightarrow$
$\qquad\qquad \{B \mid \forall X.([\![\mathit{\Gamma}]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \rightarrow \psi)\}$ where $X = \mathsf{FV}([\![\mathit{\Gamma}]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \rightarrow \psi) \setminus \mathit{var}$
$\qquad \mid \{D\; T_1 \cdots T_n \mid \psi\} \quad \Rightarrow$
$\qquad\quad \texttt{let } T'_k, \; = \; \mathsf{necessType}(\mathit{\Gamma} \vdash T_k\; \mathit{var})$ (for each k) $\texttt{in}$
$\qquad\qquad \{D\; T'_1 \cdots T'_n \mid \forall X.([\![\mathit{\Gamma}]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \rightarrow \psi)\}$
$\qquad\qquad\qquad$ where $X = \mathsf{FV}([\![\mathit{\Gamma}]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \rightarrow \psi) \setminus \mathit{var}$
$\qquad \mid x : T_1 \rightarrow T_2 \quad \Rightarrow$
$\qquad\quad \texttt{let } T'_1 \; = \; \mathsf{suffType}(\mathit{\Gamma} \vdash T_1, \; \mathit{var}) \texttt{ in}$
$\qquad\quad \texttt{let } T'_2 \; = \; \mathsf{necessType}((\mathit{\Gamma}; x : T_1) \vdash T_2, \; \mathit{var} \cup \{x\}) \texttt{ in}$
$\qquad\qquad x : T'_1 \rightarrow T'_2$

$\mathsf{suffType}(\mathit{\Gamma} \vdash T, \; \mathit{var}) \; =$
$\quad \texttt{match } T \texttt{ with}$
$\qquad \mid \{B \mid \psi\} \quad \Rightarrow$
$\qquad\qquad \{B \mid \exists X.([\![\mathit{\Gamma}]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \wedge \psi)\}$ where $X = \mathsf{FV}([\![\mathit{\Gamma}]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \wedge \psi) \setminus \mathit{var}$
$\qquad \mid \{D\; T_1 \cdots T_n \mid \psi\} \quad \Rightarrow$
$\qquad\quad \texttt{let } T'_k \; = \; \mathsf{suffType}(\mathit{\Gamma} \vdash T_k\; \mathit{var})$ (for each k) $\texttt{in}$
$\qquad\qquad \{D\; T_1 \cdots T_n \mid \exists X.([\![\mathit{\Gamma}]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \wedge \psi)\}$
$\qquad\qquad\qquad$ where $X = \mathsf{FV}([\![\mathit{\Gamma}]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \wedge \psi) \setminus \mathit{var}$
$\qquad \mid x : T_1 \rightarrow T_2 \quad \Rightarrow$
$\qquad\quad \texttt{let } T'_1, \; = \; \mathsf{necessType}(\mathit{\Gamma} \vdash T_1, \; \mathit{var}) \texttt{ in}$
$\qquad\quad \texttt{let } T'_2, \; = \; \mathsf{suffType}((\mathit{\Gamma}; x : T'_1) \vdash T_2, \; \mathit{var} \cup \{x\})\texttt{in}$
$\qquad\qquad x : T'_1 \rightarrow T'_2$

Fig. 10. The algorithm for Step 2.2

$$
\begin{aligned}
&y : \{Int \mid P_1\} \\
&\to ys : \{\texttt{List}\langle \lambda x \lambda y.x \ \leq y\rangle\ \{Int \mid P_2\} \mid P_3\} \\
&\to \{\texttt{List}\langle \lambda x \lambda y.x \ \leq y\rangle\ \{Int \mid P_4\} \mid P_5\}
\end{aligned}
$$

$$
\begin{aligned}
&P_1 \equiv \exists x, xs, z.([\![\Gamma_1]\!]_{\{x\}} \wedge \nu = x),\ \ P_2 \equiv \exists x, xs, z, l.([\![\Gamma_2]\!]_{\{x,y\}} \wedge x \leq \nu), \\
&P_3 \equiv \exists x, xs, z, l.([\![\Gamma_2]\!]_{\{xs,y\}} \wedge \texttt{len}\ \nu = \texttt{len}\ xs \wedge \texttt{elems}\ \nu = \texttt{elems}\ xs), \\
&P_4 \equiv \forall x, xs, z, l.([\![\Gamma_3]\!]_{\{y,ys\}} \Rightarrow \texttt{True}) \\
&P_5 \equiv \forall x, xs, z, l.([\![\Gamma_3]\!]_{\{l,y,ys\}} \Rightarrow \texttt{len}\ \nu = \texttt{len}\ l \wedge \texttt{elems}\ \nu = \texttt{elems}\ l) \\
&\textit{where} \\
&\Gamma_1 \equiv \Gamma,\ \ \Gamma_2 \equiv \Gamma;\ y : \{Int \mid \nu = x\} \\
&\Gamma_3 \equiv \Gamma_2;\ ys : \{\texttt{List}\langle \lambda x \lambda y.x \leq y\rangle\ \{Int \mid \nu \leq x\} \mid \\
&\qquad\qquad\qquad \texttt{len}\ \nu = \texttt{len}\ xs \wedge \texttt{elems}\ \nu = \texttt{elems}\ xs\}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\Gamma_1]\!]_{\{x\}} \equiv\ \ z = l \wedge\ \texttt{len}\ xs + 1 = \texttt{len}\ z \wedge \texttt{elems}\ xs + [x] = \texttt{elems}\ z \\
&[\![\Gamma_2]\!]_{\{x,y\}} \equiv [\![\Gamma_2]\!]_{\{xs,y\}} \equiv \\
&\qquad z = l \wedge\ \texttt{len}\ xs + 1 = \texttt{len}\ z \wedge \texttt{elems}\ xs + [x] = \texttt{elems}\ z \wedge y = x \\
&[\![\Gamma_3]\!]_{\{y,ys\}} \equiv [\![\Gamma_3]\!]_{\{l,y,ys\}} \equiv \\
&\qquad z = l \wedge\ \texttt{len}\ xs + 1 = \texttt{len}\ z \wedge \texttt{elems}\ xs + [x] = \texttt{elems}\ z \wedge y = x \\
&\qquad \wedge \texttt{len}\ ys = \texttt{len}\ xs \wedge \texttt{elems}\ ys = \texttt{elems}\ xs
\end{aligned}
$$

Fig. 11. An example output of Step 2.2

in T_2 or negatively (resp. positively) in T_1. Thus, it suffices to replace each existential formula ψ with a quantifier-free formula ψ' weaker than ψ (i.e., $\psi \Rightarrow \psi'$), and each universal formula ψ with a quantifier-free formula ψ' stronger than ψ. We discuss two procedures below.

The first procedure, which is naive but was adopted in our implementation and effective in the experiments reported in Section 4, just propagates equality information so that quantified variables are removed as much as possible. Given an existentially-quantified formula $\exists \widetilde{x}.(\psi_1 \wedge \cdots \wedge \psi_\ell)$, we collect the subset of $\{\psi, \ldots, \psi_\ell\}$ consisting of equality constraints, orient the equations (so that terms containing quantified variables tend to be replaced by those that do not contain quantified variables), and rewrite each ψ_i to ψ'_i using the equations. We then collect the subset $\{\psi'_i\}_{i \in I}$ of $\{\psi'_1, \ldots, \psi'_k\}$ that do not contain quantified variables, and replace $\exists \widetilde{x}.(\psi_1 \wedge \cdots \wedge \psi_\ell)$ with $\wedge_{i \in I} \psi'_i$. Similarly, given a universally quantified formula $\forall \widetilde{x}.(\psi_1 \wedge \cdots \wedge \psi_k \Rightarrow \psi)$, we rewrite ψ by using the equality constraints in $\psi_1, \ldots, \psi_k$. If the resulting formula ψ' contains no quantified variables, we return ψ'; otherwise the whole formula is replaced by $\perp$.

Example 3. We continue Example 2. The type obtained in Step 2.2 is shown in Figure 11. Here,

$$
\begin{aligned}
&P_5 \equiv \forall\ x, xs, z, l. \\
&\qquad (z = l\ \wedge \texttt{len}\ xs + 1 = \texttt{len}\ z\ \wedge \texttt{elems}\ xs + [x] = \texttt{elems}\ z\ \wedge x = y \\
&\qquad\ \ \wedge \texttt{len}\ ys = \texttt{len}\ xs\ \wedge \texttt{elems}\ ys = \texttt{elems}\ xs \\
&\qquad\ \ \Rightarrow \quad \texttt{len}\ \nu = \texttt{len}\ l \wedge \texttt{elems}\ \nu = \texttt{elems}\ l\ )
\end{aligned}
$$

Using the equations on the lefthand side of $\Rightarrow$, the righthand side can be rewritten as follows.

$$\begin{aligned}
&\texttt{len}\ \nu = \texttt{len}\ l \wedge \texttt{elems}\ \nu = \texttt{elems}\ l \\
&\rightsquigarrow \texttt{len}\ \nu = \texttt{len}\ z \wedge \texttt{elems}\ \nu = \texttt{elems}\ z \qquad\qquad (\text{by } z = l) \\
&\rightsquigarrow \texttt{len}\ \nu = \texttt{len}\ xs + 1 \wedge \texttt{elems}\ \nu = \texttt{elems}\ xs + [x] \\
&\qquad\qquad (\text{by } \texttt{len}\ xs + 1 = \texttt{len}\ z, \texttt{elems}\ xs + [x] = \texttt{elems}\ z) \\
&\rightsquigarrow \texttt{len}\ \nu = \texttt{len}\ ys + 1 \wedge \texttt{elems}\ \nu = \texttt{elems}\ ys + [y] \\
&\qquad\qquad (\text{by } x = y, \texttt{len}\ ys = \texttt{len}\ xs, \texttt{elems}\ ys = \texttt{elems}\ xs)
\end{aligned}$$

Since the resulting formula does not contain quantified variables, we obtain $\texttt{len}\ \nu = \texttt{len}\ ys + 1 \wedge \texttt{elems}\ \nu = \texttt{elems}\ ys + [y]$ as a sound approximation of P_5. We can eliminate quantifiers from $P_1, \ldots, P_4$ in a similar manner, and obtain the following type for auxiliary function $\Box_2$.

$$\begin{aligned}
&\Box_2 :: y : \mathit{Int} \to ys : \texttt{List}\langle \lambda x \lambda y.x \ \le y\rangle\ \mathit{Int} \to \\
&\qquad \{\texttt{List}\langle \lambda x \lambda y.x \ \le y\rangle\ \mathit{Int} \mid \texttt{len}\ \nu = \texttt{len}\ ys + 1 \wedge \texttt{elems}\ \nu = \texttt{elems}\ ys + [y]\}
\end{aligned}$$

□

Though the naive algorithm above may be effective for formulas consisting of equality constraints, it is not so for formulas containing other constraints. For example, $\exists y.(\texttt{len}\,x \le 1 + \texttt{len}\,y \wedge 2 \times \texttt{len}\,y \le z)$ is equivalent to $2 \times \texttt{len}\,x \le 2 + z$, but the naive algorithm obviously fails to output it, as there is no equality information available. The second method we discuss below first eliminates uninterpreted function symbols, and then applies quantifier elimination to the formula without uninterpreted function symbols. Consider the following formula (which is a twisted version of the formula above):

$$\exists y, w.(\texttt{len}\,x \le 1 + \texttt{len}\,y \wedge y = w \wedge 2 \times \texttt{len}\,w \le z).$$

We first pick equality constraints; $y = w$ in the case above. For each equality constraint $v_1 = v_2$, we add equalities of the form

$$E[v_1] = E[v_2]$$

whenever the term $E[v_1]$ or $E[v_2]$ occurs in the formula. In the example above, we obtain

$$\exists y, w.(\texttt{len}\,x \le 1 + \texttt{len}\,y \wedge y = w \wedge 2 \times \texttt{len}\,w \le z \wedge \texttt{len}\,y = \texttt{len}\,w)).$$

We then replace each term t constructed by uninterpreted function symbols with a fresh variable v_t.

$$\exists y, w, v_{\texttt{len}\,y}, v_{\texttt{len}\,w}.(v_{\texttt{len}\,x} \le 1 + v_{\texttt{len}\,y} \wedge y = w \wedge 2 \times v_{\texttt{len}\,w} \le z \wedge v_{\texttt{len}\,y} = v_{\texttt{len}\,w}).$$

Note that the resulting formula is weaker than the original formula, because we have lost correlations between, e.g., x and $v_{\texttt{len}\,x}$. In general, an existential formula (a universal formula, resp.) may be replaced by a weaker (a stronger,

```
infer_aux_types(Γ ⊢ t□ :: T){
    {Γ1 ⊢ □1 :: T1, ..., Γk ⊢ □k :: Tk} ← step2.1(Γ ⊢ t□ :: T);
    foreach Γi ⊢ □i :: Ti do {
        T'i ← step2.2(Γi ⊢ Ti);
        T□i ← step2.3(T'i)};
    return {□1 : T□1, ..., □k : T□k};
}
```

Fig. 12. Step 2

resp.) formula, but this is what we need for the soundness of our quantifier elimination. In the example above, we can now apply quantifier elimination for linear integer arithmetic, and obtain $2 \times v_{\mathtt{len}\,x} \leq 2 + z$. Finally, by recovering terms containing uninterpreted function symbols, we obtain $2 \times \mathtt{len}\,x \leq 2 + z$, as required. This approach would be effective in particular when the underlying logic is a logic L extended with uninterpreted function symbols, such that a complete quantifier elimination procedure exists for L.

Soundness of Step 2. The whole procedure for Step 2 is summarized in Figure 12; `step-2.3` is one of the sound but incomplete quantifier procedures discussed above. Theorem 3 below states soundness of the procedure. The first property states that the inferred types are closed (so that they can be passed to Synquid), and the second one implies that if we can find auxiliary functions of the inferred types, we can obtain a target function of type T by filling the template t with the auxiliary functions.

Theorem 3. *Given* $\{\Box_i : T_{\Box_i}\} = \mathsf{infer\_aux\_types}(\Gamma \vdash t :: T)$*, the following properties hold.*

1. $\mathsf{FV}(T_{\Box_i}) = \emptyset$
2. $(\Gamma; \Box_i : T_{\Box_i}) \vdash t :: T$

Proof. See Appendix B.

3.3 Step 3: Synthesizing auxiliary function using Synquid

Finally, we pass to Synquid the types of auxiliary functions inferred in Step 2 (Section 3.2). By filling the template with the auxiliary functions, we obtain a required target function. If Synquid fails to discover auxiliary functions (this can happen either if the types inferred in Step 2 are not inhabited by any programs, or if they are inhabited but Synquid is not powerful enough to find inhabitants), we go back to Step 1 and try another template.

$$
\begin{aligned}
f = \lambda l.\ & \texttt{match}\ l\ \texttt{with} \\
& \ \texttt{Nil} \mapsto \Box_1 \\
& \mid \texttt{Cons}\ x\ \texttt{Nil} \mapsto \Box_2\ x \\
& \mid \texttt{Cons}\ x\ xs \mapsto (\texttt{match}\ (\Box_3\ l)\ \texttt{with}\ \texttt{Pair}\ l_1\ l_2 \mapsto \mathit{append}\ (f\ l_1)\ (f\ l_2))
\end{aligned}
$$

Fig. 13. An invalid divide-and-conquer template

3.4 Limitations

Our procedure for program synthesis may fail for various reasons, due to limitations of each step. First, the syntax of templates in Figure 8 is rather restricted. For example, consider another divide-conquer template shown in Figure 13, which is obtained by replacing *split* of the divide-and-conquer template in Section 3.1 with a hole, and instead instantiating $\Box_3$ to the append function. This template is invalid due to the position in which $\Box_3$ occurs; if it were valid, we would be able to obtain a quick sort function, by instantiating $\Box_3$ with the partition function. Unfortunately allowing this (invalid) template is problematic for type inference in Step 2.1. A problem is that, in order to conclude that the subterm $\mathit{append}\ (f\ l_1)\ (f\ l_2)$ returns a sorted list, we need to infer that all the elements of l_1 are no greater than those of l_2. It is not clear at all how to infer such information from the specification of f.

The other sources of failures of our program synthesis include the incompleteness of the quantifier elimination procedure in Step 2.3, and limitations of the backend tool Synquid used in Step 3.

4 Implementation and Experiments

We have implemented a prototype program synthesis tool based on our method. The tool is written in OCaml and uses Synquid [8, 9] for the final step of our method.

We have run our tool and compared it with Synquid for several problems of synthesizing programs that manipulate lists and binary search trees. We have checked the standard libraries of functional languages such as the list library of Haskell, and chosen, as the benchmark problems, library functions whose specifications can be expressed by refinement types and whose implementations are expected to require auxiliary functions. In all the problems, no information about auxiliary functions was given to our tool and Synquid. Our tool uses the fold-style templates and the divide-conquer template discussed in Section 3.1. The experiment was conducted on a machine with 1.8GHz Intel Core i5 (8GB of memory).

The experimental results are summarized in Table 1. The column “programs” shows the names of functions to synthesize. We briefly describe them below.

Table 1. Experimental results (times are in seconds).

programs	our method total	our method type-infer	our method synquid	SYNQUID total	SYNQUID + foldr total
`list-intersect`	1.290	0.166	1.103	-	-
`list-sub`	0.603	0.110	0.478	-	-
`list-to-bst`	1.934	0.059	1.860	-	-
`list-sort`	0.910	0.105	0.791	-	3.931
`list-reverse`	0.574	0.104	0.457	-	-
`list-unique`	0.568	0.101	0.455	-	2.937
`list-concat`	0.466	0.052	0.400	-	-
`bst-to-list`	2.752	0.091	2.644	-	-
`list-mergeSort`	5.865	0.207	5.655	-	N/A

- `list-intersect`: given two sets (represented as lists), returns the intersection
- `list-sub`: given two sets (represented as lists), returns the difference
- `list-to-bst`: converts a list to a binary search tree.
- `list-sort`: sorts a list.
- `list-reverse`: reverses a list.
- `list-unique`: removes duplicate elements in a list.
- `list-concat`: flattens a list of lists.
- `bst-to-list`: converts a binary search tree to a list.
- `list-mergeSort`: sorts a list; the divide-conquer pattern is used as the default template.

The fold-style template was used as the default template, except for the last one. The three sub-columns in the column "our method" respectively show the total execution time, the time spent for the inference of the types of auxiliary functions (in Steps 1 and 2 in Section 3), and the time spent by SYNQUID (in Step 3 in Section 3). The cell "-" represents a failure. The column "SYNQUID" shows the result of running SYNQUID with no hints, and "SYNQUID+foldr" shows the result of running SYNQUID with the type of the fold-right function (shown in Figure 14) as a hint (so that SYNQUID can use the fold-right function in the target functions). The latter is based on the method for discovering auxiliary functions as proposed by Polikarpova [9]. The result "N/A" for `list-mergeSort` means "non-applicable"; given the type of the fold-right function, SYNQUID synthesizes an insertion sort program instead of a merge sort program.

As the table shows, our tool could successfully synthesize all the programs. In contrast, SYNQUID could synthesize none of the benchmark programs; it is as expected, because the benchmark programs require auxiliary functions. It may come as a surprise that, even given the type of the fold-right function, SYNQUID could synthesize only two of the benchmark programs. This is because of the limitation that the full behavior of the fold-right function is not expressed by its type, The type in Figure 14 is quite general: roughly, it describes that, for any predicate p on a list of elements of type β and a value of type γ, $foldr\ f\ seed\ ys$

$$
\begin{array}{l}
foldr :: \langle p :: \texttt{List}\ \beta \to \gamma \to \texttt{Bool}\rangle. \\
\quad f : (t : \texttt{List}\ \beta\ \to\ h : \beta\ \to\ acc : \{\gamma \mid p\ t\ \nu\} \to \{\gamma \mid p\ (\texttt{Cons}\ h\ t)\ \nu\}) \\
\quad \to seed : \{\gamma \mid p\ \texttt{Nil}\ \nu\}\ \to ys : \texttt{List}\ \beta \to\ \{\gamma \mid p\ ys\ \nu\}
\end{array}
$$

Fig. 14. The type of the fold-right function [9]

returns a value r such that $p\ ys\ r$, provided that p `Nil` *seed* holds and the accumulation function f preserves the invariant p between an input list and the corresponding output. The type still fails to describe certain information about the behavior of fold-right; for example, the type of the first argument f does not directly express the relationship between the accumulation parameter *acc* and the return value.

5 Related Work

We have already discussed the work of Polikarpova et al. [9], which we have extended to enable synthesis of programs with auxiliary functions. There are other studies of automated synthesis of functional programs [2, 6, 7, 9, 10], but we are not aware of previous methods that can automatically synthesize auxiliary functions from the specification of a main function alone. Kneuss et al. [6] discuss the synthesis of a merge sort function from a user-supplied template similar to our divide-and-conquer template, but they also require that the specification of the auxiliary function "merge" be provided by a user.

To express precise specifications of target functions, we have borrowed the type system of Polikarpova et al. [9], which is in turn based on Vazou et al.'s type system with abstract refinement types [11].

In the context of automated theorem proving, there have been studies on techniques for automated discovery of lemmas [1, 5]. Through the Curry-Howard correspondence between proofs and programs, lemmas correspond to auxiliary functions; thus, we plan to investigate the techniques for lemma discovery to refine our method.

6 Conclusion

We have proposed a method for automatically synthesizing functional programs that require auxiliary functions. We have implemented a prototype synthesis tool that uses Synquid as a backend, and confirmed that it is able to synthesize several functions with auxiliary functions. Overcoming the limitations discussed in Section 3.4 is left for future work.

Acknowledgments We would like to thank anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP15H05706 and JP16K16004.

References

1. Aoto, T.: Sound lemma generation for proving inductive validity of equations. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11, 2008, Bangalore, India. LIPIcs, vol. 2, pp. 13–24. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2008)
2. Frankle, J., Osera, P., Walker, D., Zdancewic, S.: Example-directed synthesis: a type-theoretic interpretation. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 802–815. ACM (2016)
3. Gulwani, S., Harris, W.R., Singh, R.: Spreadsheet data manipulation using examples. Communications of the ACM **55**(8), 97–105 (2012)
4. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 62–73. ACM (2011)
5. Kapur, D., Subramaniam, M.: Lemma discovery in automated induction. In: McRobbie, M.A., Slaney, J.K. (eds.) Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings. Lecture Notes in Computer Science, vol. 1104, pp. 538–552. Springer (1996)
6. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: Hosking, A.L., Eugster, P.T., Lopes, C.V. (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. ACM (2013)
7. Osera, P., Zdancewic, S.: Type-and-example-directed program synthesis. In: Grove, D., Blackburn, S. (eds.) Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015. pp. 619–630. ACM (2015)
8. Polikarpova, N.: Synquid. `https://bitbucket.org/nadiapolikarpova/synquid/`.
9. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Krintz, C., Berger, E. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 522–538. ACM (2016), `http://doi.acm.org/10.1145/2908080`
10. Solar-Lezama, A.: Program synthesis by sketching. Ph.D. thesis, University of California, Berkeley (2008)
11. Vazou, N., Rondon, P.M., Jhala, R.: Abstract refinement types. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 209–228. Springer (2013)

Appendix

A Well-formedness of Types and Type Environments

A formula ψ is *well formed* in the environment Γ, written $\Gamma \vdash \psi$, when it has a boolean sort under the assumption that each free variable in ψ has the sort declared in Γ.

The well-formedness relations on types and type environments, $\Gamma \vdash T$ and $\vdash \Gamma$ respectively, are defined by the rules given below.

$$\frac{\Gamma; \nu : B \vdash \psi}{\Gamma \vdash \{B \mid \psi\}} \ (\textsc{WfT-Sc}) \qquad\qquad \frac{\Gamma; C \vdash T}{\Gamma \vdash \texttt{let}\ C\ \texttt{in}\ T} \ (\textsc{WfT-Ctx})$$

$$\frac{\Gamma \vdash \{B \mid \psi\} \qquad \Gamma; x : \{B \mid \psi\} \vdash T}{\Gamma \vdash x : \{B \mid \psi\} \to T} \ (\textsc{WfT-Fun1})$$

$$\frac{T_x \text{ is not of the form } \{B \mid \psi\} \qquad \Gamma \vdash T_x \qquad \Gamma \vdash T}{\Gamma \vdash x : T_x \to T} \ (\textsc{WfT-Fun2})$$

$$\frac{}{\vdash \emptyset} \ (\textsc{WfTE-Emp})$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash T \qquad x \text{ does not occur in } \Gamma}{\vdash \Gamma; x : T} \ (\textsc{WfTE-T})$$

$$\frac{\vdash \Gamma \qquad \Gamma \vdash \psi}{\vdash \Gamma; \psi} \ (\textsc{WfTE-P})$$

B Proof

In this section, we will prove Theorem 1, Theorem 2 and Theorem 3.

B.1 Useful Lemmas

Hereafter we implicitly assume that for every environment Γ, $\Gamma = (\Gamma_1; x : T; \Gamma_2; y : U; \Gamma_3)$ implies $x \neq y$, by renaming if necessarily.

Lemma 1.

- *Let Γ be an environment and Γ' be a permutation. Assume $\Gamma \vdash T <: U$ and Γ' is well-formed (i.e. $\Gamma' \vdash$). Then $\Gamma' \vdash T <: U$.*
- *Assume that $\Gamma, C, \Delta \vdash T <: U$ and that the judgement $\Gamma, \Delta \vdash T <: U$ is well-formed (i.e. $\Gamma, \Delta \vdash T$ and $\Gamma, \Delta \vdash U$ and thus $\Gamma, \Delta \vdash$). Then $\Gamma, \Delta \vdash T <: U$.*

In particular, if T is a function type, $\Gamma;\, x\colon T; \Delta \vdash U <: U'$ implies $\Gamma; \Delta \vdash U <: U'$.

For a given type environment Γ, we write $\mathrm{dom}(\Gamma)$ for the set of variables declared in Γ. Formally

$$\begin{aligned}\mathrm{dom}(\emptyset) &:= \emptyset\\ \mathrm{dom}(\Gamma;\, x\colon T) &:= \mathrm{dom}(\Gamma) \cup \{x\}\\ \mathrm{dom}(\Gamma;\, \psi) &:= \mathrm{dom}(\Gamma).\end{aligned}$$

The next lemma relies on the technical assumption that $\nu \in \mathsf{FV}(\psi)$ for every scalar type $\{B \mid \psi\}$.

Lemma 2. *Let Γ be a well-formed environment and $var \subseteq \mathrm{dom}(\Gamma)$. Then*

$$[\![\Gamma]\!]_{var} \supseteq var.$$

Proof. By easy induction.

B.2 Proof of Theorem 1

Lemma 3. *Let Γ be a well-formed environment, $e_\Box$ be an E-term with a hole, T be a well-formed type (i.e. $\Gamma \vdash T$) and C be a context (with $\Gamma \vdash C$). Suppose that*

$$\{\Delta \vdash \Box_i :: U\} = \mathsf{extractConst}(\Gamma,\ e_\Box,\ T\ , C).$$

Let S be a type such that $\emptyset \vdash S$ and $\Delta \vdash S <: U$. Then there exists a well-formed contextual $\texttt{let}\ C'\ \texttt{in}\ T'$ *(i.e. $\Gamma \vdash \texttt{let}\ C'\ \texttt{in}\ T'$) such that*

$$(\Gamma;\, \Box_i : S) \vdash e_\Box :: \texttt{let}\ C'\ \texttt{in}\ T' \tag{2}$$

$$(\Gamma;\, \Box_i : S; C; C') \vdash T' <: T. \tag{3}$$

Proof. By induction on the structure of $e_\Box$.

- (Case: $e_\Box = \Box_i$) By definition (Fig. 9),

  $$\mathsf{extractConst}(\Gamma,\ \Box_i,\ T\ , C) = \{\Gamma; C \vdash \Box_i :: T\}$$

  and thus $\Delta = (\Gamma; C)$ and $U = T$. Let $C' = \emptyset$ and $T' = S$. By the variable rule,

  $$(\Gamma;\, \Box_i : S) \vdash \Box_i :: \texttt{let}\ \emptyset\ \texttt{in}\ S$$

  and thus we obtain (2). By the assumption,

  $$(\Gamma; C) \vdash S <: T$$

  which implies (3) by weakening.

– (Case: $e_\Box = e'_\Box\, e_0$) Let $\texttt{let}\ C_0\ \texttt{in}\ T_0$ be a (well-formed) contextual type found by the procedure. Then

$$\Gamma \vdash e_0 :: \texttt{let}\ C_0\ \texttt{in}\ T_0$$

and

$$\begin{aligned}\{\Box_i : (\Delta \vdash * <: U)\} &= \mathsf{extractConst}(\Gamma,\ e'_\Box\, e_0,\ T\ , C)\\ &= \mathsf{extractConst}(\Gamma,\ e'_\Box,\ y : T_0 \to T\ , C; C_0)\end{aligned}$$

where y is a fresh variable not appearing in T. By the induction hypothesis, there exists a well-formed contextual type $\Gamma \vdash \texttt{let}\ C'_0\ \texttt{in}\ T'_1$ such that

$$\begin{aligned}&(\Gamma;\ \Box_i : S) \vdash e'_\Box :: \texttt{let}\ C'_0\ \texttt{in}\ T'_1\\ &(\Gamma;\ \Box_i : S; C; C_0; C'_0) \vdash T'_1 <: (y : T_0 \to T).\end{aligned}$$

By the second condition, $T'_1 = (y : T'_0 \to T')$ for some y, T'_0 and T' and

$$\begin{aligned}&(\Gamma;\ \Box_i : S; C; C_0; C'_0) \vdash T_0 <: T'_0\\ &(\Gamma;\ \Box_i : S; C; C_0; C'_0; y : T_0) \vdash T' <: T.\end{aligned} \tag{4}$$

Let $C' = (C'_0; C_0; y : T_0)$, which is well-formed (i.e. $\Gamma \vdash C'$) since $\Gamma \vdash C'_0$ and $\Gamma \vdash \texttt{let}\ C_0\ \texttt{in}\ T_0$. Since C'_0 does not depend on C_0, one can exchange C_0 and C'_0 in (4), which leads to (3). Since $\Gamma; C_0 \vdash T_0$ and $\Gamma; C'_0 \vdash T'_0$, by Lemma 1, we can drop the context C from the former judgement and obtain

$$(\Gamma;\ \Box_i : S; C'_0; C_0) \vdash T_0 <: T'_0.$$

Therefore

$$\frac{\begin{array}{c}(\Gamma;\ \Box_i : S) \vdash e'_\Box :: \texttt{let}\ C'_0\ \texttt{in}\ (y : T'_0 \to T')\\ (\Gamma;\ \Box_i : S; C'_0) \vdash e :: \texttt{let}\ C_0\ \texttt{in}\ T_0\\ (\Gamma; C'_0; C_0) \vdash T_0 <: T'_0\end{array}}{(\Gamma;\ \Box_i : S) \vdash e'_\Box\ e :: \texttt{let}\ C'_0; C_0; y : T_0\ \texttt{in}\ T'}\ (\mathrm{APP})$$

and we have (2) as desired.

– (Case: $e_\Box = e'_\Box\ f$) Similar to the above case.

$\Box$

Lemma 4. *Let Γ be a well-formed environment, $e_\Box$ be an E-term with a hole, T be a well-formed type (i.e. $\Gamma \vdash T$). Suppose that*

$$\{\Delta \vdash \Box_i :: U\} = \mathsf{extractConst}(\Gamma,\ e_\Box,\ T\ , \emptyset).$$

Let S be a type such that $\emptyset \vdash S$ and $\Delta \vdash S <: U$. Then

$$(\Gamma;\ \Box_i : S) \vdash e_\Box :: T.$$

Proof. By Lemma 3, there exists a well-formed contextual type $\texttt{let}\ C'\ \texttt{in}\ T'$ such that

$$(\Gamma; \square_i : S) \vdash e_\square :: \texttt{let}\ C'\ \texttt{in}\ T'$$

and

$$(\Gamma; \square_i : S; C') \vdash T' <: T.$$

By the subtyping rule,

$$\frac{(\Gamma; \square_i : S) \vdash e_\square :: \texttt{let}\ C'\ \texttt{in}\ T' \qquad (\Gamma; \square_i : S; C') \vdash T' <: T}{(\Gamma; \square_i : S) \vdash e_\square :: T}\ (\text{SUBT})$$

as required. $\square$

Proof (of Theorem 1). By induction on the structure of t. The base case $t = e_\square$ is a consequence of Lemma 4. Other cases are easy. $\square$

B.3 Proof of Theorem 2

By induction on the construction of the type T.

– (Case: $T = \{B \mid \psi\}$ with $B = \mathit{Bool}$ or Int) Let

$$S := \mathsf{necessType}(\Gamma \vdash T,\ \mathit{var})$$
$$S' := \mathsf{suffType}(\Gamma \vdash T,\ \mathit{var}).$$

By definition (Fig. 10), we have

$$S = \{B \mid \forall x_1 \dots x_n.([\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \to \psi)\}$$
$$S' = \{B \mid \exists x_1 \dots x_n.([\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \wedge \psi)\}$$

where $\{x_1, \dots, x_n\} = (\mathsf{FV}([\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}}) \cup \mathsf{FV}(\psi)) \setminus (\mathit{var} \cup \{\nu\})$. Let $\boldsymbol{x}$ be the sequence $x_1 \dots x_n$ of variables.
Obviously $\mathsf{FV}(S) = \mathsf{FV}(S') \subseteq \mathit{var}$.
We prove $\Gamma \vdash S <: T$. By the subtyping rule, it suffices to show that $\Gamma \vdash B <: B$, which trivially holds, and

$$[\![\Gamma]\!]_{\mathsf{FV}((\forall \boldsymbol{x}.\ [\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \Rightarrow \psi) \Rightarrow \psi)} \wedge \big(\forall \boldsymbol{x}.\ [\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \Rightarrow \psi\big) \Rightarrow \psi$$

is valid. By definition,

$$\mathsf{FV}((\forall \boldsymbol{x}.\ [\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \Rightarrow \psi) \Rightarrow \psi) = \mathsf{FV}(\forall \boldsymbol{x}.\ [\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \Rightarrow \psi) \cup \mathsf{FV}(\psi).$$

By Lemma 2, $\mathsf{FV}([\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}}) \supseteq \mathit{var}$ and thus

$$\mathsf{FV}(\forall \boldsymbol{x}.\ [\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \Rightarrow \psi) \cup \mathsf{FV}(\psi) = \mathit{var} \cup \mathsf{FV}(\psi).$$

Hence the above formula is equivalent to

$$[\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \wedge \big(\forall \boldsymbol{x}.\ [\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup \mathit{var}} \Rightarrow \psi\big) \Rightarrow \psi,$$

which is easy to show.
We prove $\Gamma \vdash S' <: T$. By the subtyping rule, it suffices to show that $\Gamma \vdash B <: B$ and

$$[\![\Gamma]\!]_{\mathsf{FV}(\exists \boldsymbol{x}.\, [\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup var} \wedge \psi)} \wedge \psi \Rightarrow \big(\exists \boldsymbol{x}.\, [\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup var} \wedge \psi\big).$$

By the same argument as above, this is equivalent to

$$[\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup var} \wedge \psi \Rightarrow \big(\exists \boldsymbol{x}.\, [\![\Gamma]\!]_{\mathsf{FV}(\psi)\cup var} \wedge \psi\big),$$

which is true.

- (Case: $T = \{D\, T_1\, \cdots\, T_n \mid \psi\}$) Let

$$\begin{aligned}\{D\, S_1\, \ldots\, S_n \mid \varphi\} &= \mathsf{necessType}(\Gamma \vdash \{D\, T_1\, \ldots\, T_n \mid \psi\},\ var)\\ \{D\, S'_1\, \ldots\, S'_n \mid \varphi'\} &= \mathsf{suffType}(\Gamma \vdash \{D\, T_1\, \ldots\, T_n \mid \psi\},\ var).\end{aligned}$$

By definition, for each $i \leq n$,

$$\begin{aligned}S_i &= \mathsf{necessType}(\Gamma \vdash T_i,\ var)\\ S'_i &= \mathsf{suffType}(\Gamma \vdash T_i,\ var).\end{aligned}$$

Since the shapes of the formulas φ and φ' are similar to the above case, we can prove that both

$$[\![\Gamma]\!]_{\mathsf{FV}(\varphi \Rightarrow \psi)} \wedge \varphi \Rightarrow \psi$$

and

$$[\![\Gamma]\!]_{\mathsf{FV}(\psi \Rightarrow \varphi')} \wedge \psi \Rightarrow \varphi'$$

are valid.
To prove $\Gamma \vdash S <: T$, it suffices to show that

$$\Gamma \vdash S_i <: T_i$$

for every i, which follows from the induction hypothesis. The subtyping judgement $\Gamma \vdash T <: S'$ can be proved similarly.
$\mathsf{FV}(S) \subseteq var$ follows from the induction hypothesis $\mathsf{FV}(S_i) \subseteq var$ and $\mathsf{FV}(\varphi) \subseteq var$, which follows from the construction. The proof of $\mathsf{FV}(S') \subseteq var$ is similar.

- (Case: $T = x : T_1 \to T_2$) Let

$$\begin{aligned}S &:= \mathsf{necessType}(\Gamma \vdash T,\ var)\\ S' &:= \mathsf{suffType}(\Gamma \vdash T,\ var).\end{aligned}$$

By definition,

$$\begin{aligned}S &= (x : S_1 \to S_2)\\ S' &= (x : S'_1 \to S'_2)\end{aligned}$$

and

$$\begin{aligned}
S_1 &= \mathsf{suffType}(\Gamma \vdash T_1,\ var)\\
S_2 &= \mathsf{necessType}(\Gamma \vdash T_2,\ var \cup \{x\})\\
S_1' &= \mathsf{necessType}(\Gamma \vdash T_1,\ var)\\
S_2' &= \mathsf{suffType}(\Gamma \vdash T_2,\ var \cup \{x\}).
\end{aligned}$$

By the induction hypothesis and the definition of free variables, $\mathsf{FV}(S) \subseteq var$ and $\mathsf{FV}(S') \subseteq var$.
The subtyping judgement $\Gamma \vdash S <: T$ follows from the induction hypotheses $\Gamma \vdash T_1 <: S_1$ and $\Gamma;\, x\colon T_1 \vdash S_2 <: T_2$. Similarly $\Gamma \vdash T <: S'$ holds.

B.4 Proof of Theorem 3

Theorem 3 is derived from Theorem 1, Theorem 2 and the soundness of step 2.3 discussed in Section 3.2.