Modular Verification of Higher-order Functional Programs

Ryosuke Sato and Naoki Kobayashi

The University of Tokyo
{ryosuke,koba}@kb.is.s.u-tokyo.ac.jp

Abstract. Fully automated verification methods for higher-order functional programs have recently been proposed based on higher-order model checking and/or refinement type inference. Most of those methods are, however, whole program analyses, suffering from the scalability problem. To address the problem, we propose a modular method for fully automated verification of higher-order programs. Our method takes a program consisting of multiple top-level functions as an input, and repeatedly applies procedures for (i) guessing refinement intersection types of each function in a counterexample-guided manner, and (ii) checking that each function indeed has the guessed refinement intersection types, until the whole program is proved/disproved to be safe. To avoid the whole program analysis, we introduce the notion of *modular counterexamples*, and utilize them in (i), and employ Sato et al.'s technique of reducing refinement type checking to assertion checking in (ii). We have implemented the proposed method as an extension to MoCHi, and confirmed its effectiveness through experiments.

1 Introduction

Thanks to the recent advance in higher-order model checking and refinement type inference, various methods and tools for automated verification of functional programs have been proposed recently [9, 15, 21, 20, 12]. For example, MoCHi [9, 18], a software model checker for functional programs, statically checks whether a given program may fail due to run-time errors such as assertion failures, uncaught exceptions, and pattern match failures, in a fully automatic manner. It outputs refinement (intersection) types as certificates of the safety if the program does not fail, and outputs a concrete execution path that causes a run-time error otherwise. Most of the *fully* automated verification methods proposed so far are whole program analyses, suffering from the scalability problem. (On the other hand, *semi*-automated methods that rely on users' annotations on invariants usually work in a compositional manner [6, 16, 11, 19, 28].)

To address the scalability problem, we propose a modular verification method for higher-order functional programs, which utilizes an existing software model checker for functional programs as a backend. An input for the verification method is a pair consisting of (i) a program P of the form:

$$\textbf{let rec } f_1\ \widetilde{x}_1 = t_1 \textbf{ in } \cdots \textbf{ let rec } f_n\ \widetilde{x}_n = t_n \textbf{ in } f_n$$

(which is abbreviated as $\langle f_1\,\widetilde{x} = t_1, \ldots, f_n\,\widetilde{x} = t_n\rangle$) and (ii) a refinement type specification τ. Here, $f_1, \ldots, f_i$ may occur in t_i.[1] Each function definition $f_i\ \widetilde{x}_i = t_i$, which may contain local function definitions, is treated as a "module", i.e., the unit of verification in our modular verification method. The goal is to check whether $\models P : \tau$, i.e., whether P has (semantically) type τ (which entails that P does not fail; for example, P has type $\mathbf{int} \to \mathbf{int}$ only if, for every integer n, $P\ n$ does not fail, and either returns an integer or diverges).

Our method infers refinement types of each function by using the following two components:

- `typeSynthesizer`, which generates a candidate refinement type environment $f_1 : \sigma_1, \ldots, f_n : \sigma_n$ (which maps each f_i to the *set* σ_i of types) such that $\tau \in \sigma_n$ from the type checking problem $\overset{?}{\models} \langle f_1\,\widetilde{x}_1 = t_1, \ldots, f_n\,\widetilde{x}_n = t_n\rangle : \tau$ and *modular counterexamples* (which will be explained later).
- `typeChecker`, which checks whether

$$f_1 : \sigma_1, \ldots, f_{k-1} : \sigma_{k-1} \models \mathbf{fix}(f_k, \lambda\widetilde{x}_k.\, t_k) : \tau_k$$

  holds (where $\mathbf{fix}(f_k, \lambda\widetilde{x}_k.\, t_k)$ denotes the recursive function defined by $f_k\,\widetilde{x}_k = t_k$), given a refinement type environment $f_1 : \sigma_1, \ldots, f_{k-1} : \sigma_{k-1}$, a candidate τ_k of refinement type of f_k, and a function definition $f_k\,\widetilde{x}_k = t_k$ for some $k \in \{1, \ldots, n\}$, and outputs a modular counterexample if $f_1 : \sigma_1, \ldots, f_{k-1} : \sigma_{k-1} \models \mathbf{fix}(f_k, \lambda\widetilde{x}_k.\, t_k) : \tau_k$ does not hold

Figure 1 describes the overall procedure of our method, utilizing the two components mentioned above. Given a program P and a refinement type specification τ, the main function first sets (i) the type environment Γ (which keeps the set of types that have already been proved to be valid) to the empty type environment (line 2), (ii) the candidate type environment $\Gamma_{\mathtt{cand}}$ to one containing only $f_n : \{\tau\}$ (line 3), and (iii) the set of (modular) counterexamples to the empty set (line 4). The main function then calls `validateTE` (line 5). Given the current type environment Γ and the current candidate type environment $\Gamma_{\mathtt{cand}}$, the function `validateTE` checks whether each $\tau' \in \Gamma(f_i)$ is a valid type for f_i for each $i \in \{1, \ldots, n\}$, by repeatedly calling `typeChecker` (line 10). Here, $P(f_i)$ denotes $\mathbf{fix}(f_i, \lambda\widetilde{x}_i.\, t_i)$, the function defined by $f_i\ \widetilde{x}_i = t_i$. If τ' is a valid type, then it is added to the set $\Gamma(f_i)$ of valid types of f_i (line 11); otherwise the counterexample returned by `typeChecker` is added to Π (line 12). If the type τ of the whole program has been proved correct, then the verification succeeds (line 13). Otherwise, `typeSynthesizer` is called to obtain a refined candidate type environment (line 15), and `validateTE` is called again (line 16). If there is no way to refine the candidate, we can conclude that the program is untypable, i.e., does not meet the specification (line 17).

[1] Mutual recursion can be realized by passing $f_{i+1}, \ldots, f_n$ as arguments of f_i. For example, **let rec** $f_1\ x = C_1[f_1, f_2]$ **and** $f_2\ x = C_2[f_1, f_2]$ **in** f_2 can be expressed as **let rec** $f_1\ f_2\ x = C_1[f_1\ f_2, f_2]$ **in let rec** $f_2'\ x = C_2[f_1\ f_2', f_2']$ **in** f_2'.

```
1: main(P, τ) =
2:  let Γ = {f_1 : ∅, ..., f_n : ∅} in (* types that have been validated so far *)
3:  let Γ_cand = {f_1 : ∅, ..., f_{n-1} : ∅, f_n : {τ}} in (* initial type candidates *)
4:  let Π = ∅ in (* the set of counterexamples found so far *)
5:    validateTE(P, τ, Γ, Γ_cand, Π)
6:
7: validateTE(P, τ, Γ, Γ_cand, Π) =
8:  for i=1 to n do (* Check each type candidate *)
9:    {for each τ' in Γ_cand(f_i) do
10:      match typeChecker(Γ, P(f_i), τ') with
11:         OK -> add τ' to Γ(f_i)
12:       | NG(π) -> add π to Π };
13: if τ ∈ Γ(f_n) then return "yes"
14: else
15:    match typeSynthesizer(P, τ, Π) with
16:       Some(Γ'_cand) -> validateTE(P, τ, Γ, Γ'_cand, Π)
17:     | None -> return "no"
```

Fig. 1. The overall procedure

Our verification method is modular in that the (semantic) typability of each function definition is checked separately by using `typeChecker`. The component `typeSynthesizer` takes the whole program as an input, but as we describe later, it looks at only part of the program that is relevant to the set Π of modular counterexamples found so far. Thus, our new method is expected to scale to larger programs than the previous whole program analysis approach [9, 18], as confirmed by experiments.

The description above explains how to verify a single whole program in a modular manner. There is a further benefit when our modular verification method is applied to verification of multiple programs that share the same library. Suppose we have a library function $f\,x = t_1$ and two client functions $g\,y = t_2$ and $h\,z = t_3$, whose refinement type specifications are τ_2 and τ_3. In that case, we run the procedure in Fig. 1 first for $\stackrel{?}{\models} \langle f\,x = t_1, g\,y = t_2\rangle : \tau_2$. If the verification is successful, we obtain a witness type environment $f : \sigma_1, g : \sigma_2$. The information that f has types σ_1 can then be used in the verification of the other client program. For that purpose, when the procedure in Fig. 1 is called for the query $\stackrel{?}{\models} \langle f\,x = t_1, h\,z = t_3\rangle : \tau_3$, we just need to set $\Gamma(f)$ to σ_1 (instead of $\emptyset$) on the second line. If the type information $f : \sigma_1$ is sufficient, then the verification of h will succeed without re-analyzing the definition of f. Otherwise, additional types for f may be inferred by reanalyzing the definition of f, and can later be used for analyzing other client programs.

The rest of this paper is structured as follows. Section 2 introduces the target language of our verification method. Section 3 overviews our method through an example. Section 4 describes the two components. Section 5 reports an imple-

$$
\begin{aligned}
P \text{ (programs)} &::= \langle f_1\ \widetilde{x_1} = t_1, \ldots, f_n\ \widetilde{x_n} = t_n \rangle \\
t \text{ (terms)} &::= n \mid x \mid *_{\mathbf{int}} \mid \mathtt{op}(\widetilde{t}) \mid \mathbf{fix}(f, \lambda x.\, t) \mid t_1\, t_2 \\
&\quad \mid \mathbf{if}^{\ell}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3 \mid \mathbf{fail} \\
\kappa \text{ (simple types)} &::= \mathbf{int} \mid \kappa_1 \to \kappa_2
\end{aligned}
$$

Fig. 2. Syntax

mentation and experimental results. Section 6 discusses related work, and Sect. 7 concludes the paper.

2 Language

In this section, we introduce the target language of our verification method.

2.1 Syntax and Semantics

The target of our method is a simply-typed, call-by-value, higher-order functional language with recursion. Its syntax is summarized in Fig. 2.

We use the meta-variables x, y, z, f, g, ... for variables. We write $\widetilde{\cdot}$ for a sequence; for example, $\widetilde{x}$ stands for a sequence of variables. For the sake of simplicity, we consider only integers as base type values. We represent Booleans using integers, and sometimes write **true** for 1 and **false** for 0. The meta-variables n and $\mathtt{op}$ range over the sets of integers, and primitive operations on integers, respectively.

A program P is a sequence of recursive function definitions $\langle f_1\ \widetilde{x_1} = t_1, \ldots, f_n\ \widetilde{x_n} = t_n \rangle$. Here, we require that $\widetilde{x_i}$ may not be an empty sequence and f_i may occur only in $t_i, \ldots, t_n$. When $P = \langle f_1\ \widetilde{x_1} = t_1, \ldots, f_n\ \widetilde{x_n} = t_n \rangle$, we write $dom(P)$ for $\{f_1, \ldots, f_n\}$, and $P(f_i)$ for $\mathbf{fix}(f_i, \lambda \widetilde{x_i}.\, t_i)$.

The term $*_{\mathbf{int}}$ evaluates to some integer in a non-deterministic manner. We write $*_{\mathbf{bool}}$ for $*_{\mathbf{int}} \geq 0$, which represents a non-deterministic Boolean. The term $\mathtt{op}(t_1, \ldots, t_k)$ applies the operation $\mathtt{op}$ to the values of $t_1, \ldots, t_k$. We sometimes use the infix notation for a binary operation and write $x\, \mathtt{op}\, y$ for $\mathtt{op}(x, y)$. The term $\mathbf{fix}(f, \lambda x.\, t)$ denotes the recursive function defined by $f\ x = t$.[2] We write $\lambda x.\, t$ for $\mathbf{fix}(f, \lambda x.\, t)$ if f does not occur in t. The term $t_1\, t_2$ applies t_1 to t_2. We write $\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2$ for $(\lambda x.\, t_2)\, t_1$, and also write $t_1; t_2$ if x does not occur in t_2. The conditional expression $\mathbf{if}^{\ell}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3$ evaluates t_2 if the value of t_1 is non-zero and t_3 otherwise; ℓ is a label used only during verification. We assume that a unique label is assigned to each conditional expression. We omit labels

[2] Thus, whether a recursive function is introduced by a top-level function definition or by $\mathbf{fix}(f, \lambda x.\, t)$ does not matter for an execution of a program; it matters only for the modular verification method, which treats each top-level function definition as the unit of modular verification.

$$E[*_{\mathbf{int}}] \longrightarrow_P E[n]$$

$$E[\mathtt{op}(v_1, \ldots, v_k)] \longrightarrow_P E[[\![\mathtt{op}]\!](v_1, \ldots, v_k)]$$

$$E[\mathbf{fix}(f, \lambda x.\, t)\, v] \longrightarrow_P E[[v/x][\mathbf{fix}(f, \lambda x.\, t)/f]t]$$

$$\frac{v \neq 0}{E[\mathbf{if}^{\ell}\ v\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2] \longrightarrow_P E[t_1]}$$

$$\frac{v = 0}{E[\mathbf{if}^{\ell}\ v\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2] \longrightarrow_P E[t_2]}$$

$$\frac{(f\, \widetilde{x} = t) \in P}{E[f] \longrightarrow_P E[\mathbf{fix}(f, \lambda \widetilde{x}.\, t)]} \qquad \frac{E \neq [\,]}{E[\mathbf{fail}] \longrightarrow_P \mathbf{fail}}$$

$$E \text{ (evaluation contexts)} ::= [\,] \mid \mathtt{op}(\widetilde{v}, E, \widetilde{t}) \mid E\, t \mid v\, E \mid \mathbf{if}^{\ell}\ E\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2$$
$$v \text{ (values)} ::= n \mid \mathbf{fix}(f, \lambda x.\, t)$$

Fig. 3. Operational semantics of the language

when they are not important. The term **fail** aborts the execution. We write $\mathbf{assert}^{\ell}(t)$ for $\mathbf{if}^{\ell}\ t\ \mathbf{then}\ 1\ \mathbf{else}\ \mathbf{fail}$, which aborts the program if the value of t is **false** (i.e., 0). We also write $\mathbf{assume}^{\ell}\,(t)$ for $\mathbf{if}^{\ell}\ t\ \mathbf{then}\ 1\ \mathbf{else}\ \mathbf{fix}(f, \lambda x.\, f\, x)\, 0$.

We consider only programs that are well-typed in the standard simple type system; the typing rules are omitted. We write $\mathcal{K} \vdash_{\mathrm{ST}} t : \kappa$ if t has simple type κ under simple type environment $\mathcal{K}$.

The (small-step) operational semantics of the language is defined in Fig. 3. In the figure, $[\![\mathtt{op}]\!]$ is the semantic integer function denoted by $\mathtt{op}$. We write $\longrightarrow_P^*$ for the reflexive and transitive closure of $\longrightarrow_P$. We omit the subscript P when it is clear from the context.

2.2 Refinement Intersection Types

We use refinement intersection types for describing properties of programs or terms. The syntax of (refinement intersection) types is defined by:

$$\begin{aligned} \tau \text{ (refinement types)} &::= \{x : \mathbf{int} \mid \phi\} \mid (x : \sigma) \to \tau \\ \sigma \text{ (intersection types)} &::= \{\tau_1, \ldots, \tau_k\}_{\kappa} \\ \phi \text{ (refinement predicates)} &::= n \mid x \mid \mathtt{op}(\phi_1, \phi_2). \end{aligned}$$

The refinement type $\{x : \mathbf{int} \mid \phi\}$ denotes the set of integers x that satisfy the refinement predicate ϕ. For example, $\{x : \mathbf{int} \mid x \geq 0\}$ is the type of non-negative integers. We often abbreviate $\{x : \mathbf{int} \mid \mathbf{true}\}$ to $\mathbf{int}$. The intersection type

$$f_1 : \sigma_1, \ldots, f_n : \sigma_n \models^P t : \tau \quad \stackrel{\text{def}}{=}$$
$$f_1 : ST(\sigma_1), \ldots, f_n : ST(\sigma_n) \vdash_{\text{ST}} t : ST(\tau) \quad \text{and}$$
$$\models^P [v_1/f_1, \ldots, v_n/f_n]t : \tau \text{ for any } v_1, \ldots, v_n \text{ s.t. } \models^P_{\text{v},\wedge} v_i : \sigma_i \text{ for all } i \in \{1, \ldots, n\}$$
$$\models^P t : \tau \quad \stackrel{\text{def}}{=}$$
$$\vdash_{\text{ST}} t : ST(\tau),\ t \not\longrightarrow^*_P \textbf{fail}, \text{ and } \models^P_{\text{v}} v : \tau \text{ for every } v \text{ such that } t \longrightarrow^*_P v$$
$$\models^P_{\text{v}} n : \{x : \textbf{int} \mid \phi\} \quad \stackrel{\text{def}}{=} \quad \models^P_{\text{p}} [n/x]\phi$$
$$\models^P_{\text{v}} \textbf{fix}(f, \lambda x.\, t) : (y : \sigma) \to \tau \quad \stackrel{\text{def}}{=}$$
$$\models^P \textbf{fix}(f, \lambda x.\, t)\, v' : [v'/y]\tau \text{ for every } v' \text{ s.t. } \models^P_{\text{v},\wedge} v' : \sigma$$
$$\models^P_{\text{v},\wedge} v : \{\tau_1, \ldots, \tau_n\}_\kappa \quad \stackrel{\text{def}}{=} \quad \vdash_{\text{ST}} v : \kappa \quad \text{and} \quad \models^P_{\text{v}} v : \tau_i \text{ for all } i \in \{1, \ldots, n\}$$
$$\models^P_{\text{p}} \phi \quad \stackrel{\text{def}}{=} \quad v = \textbf{true} \text{ for any } v \text{ s.t. } \phi \longrightarrow^*_P v$$

Fig. 4. Semantics of types

$\{\tau_1, \ldots, \tau_k\}_\kappa$ describes values that have type τ_i whose simple type is κ for every $i \in \{1, \ldots, k\}$. We often omit the subscript κ when they are not important, and treat $\{\tau_1, \ldots, \tau_k\}_\kappa$ as a set of refinement types. The type $(x : \sigma) \to \tau$ denotes the set of functions that take an argument v of (intersection) type σ and return a value of type $[v/x]\tau$. For example, $(x : \{y : \textbf{int} \mid \textbf{true}\}) \to \{r : \textbf{int} \mid r \geq x\}$ is the type of functions that take any integer as an argument and return an integer no less than the argument. In $(x : \sigma) \to \tau$, we allow x to occur in τ only if σ is of the form $\{y : \textbf{int} \mid \phi\}$. In other words, we do not allow dependencies on function variables. We write $\sigma \to \tau$ for $(x : \sigma) \to \tau$ if x does not occur in τ.

We say that τ is a *refinement* of a simple type κ if $\tau :: \kappa$ is derivable from the following rules:

$$\{x : \textbf{int} \mid \phi\} :: \textbf{int} \qquad \frac{\sigma :: \kappa_1 \qquad \tau :: \kappa_2}{((x : \sigma) \to \tau) :: (\kappa_1 \to \kappa_2)} \qquad \frac{\tau_i :: \kappa \quad \text{for each } i \in \{1, \ldots, n\}}{\{\tau_1, \ldots, \tau_n\}_\kappa :: \kappa}$$

Henceforth, we consider only refinement types and intersection types that are refinements of some simple types. For such a refinement type τ (an intersection type σ, resp.), the simple type κ such that $\tau :: \kappa$ ($\sigma :: \kappa$, resp.) is uniquely determined. We write $ST(\tau)$ ($ST(\sigma)$, resp.) for it. We also write $ST(\Gamma)$ for $f_1 : ST(\sigma_1), \ldots, f_k : ST(\sigma_k)$ when $\Gamma = f_1 : \sigma_1, \ldots, f_k : \sigma_k$.

The semantics of types is defined in Fig. 4 using logical relations. The relation $\models^P_{\text{v}} v : \tau$ means that the value v has type τ, and $\models^P t : \tau$ means that reduction of the closed (where the top-level functions $f_1, \ldots, f_n$ are considered bound variables) term t never fails, and that every value v (if there is any) of t has type τ. The relation $\Gamma \models^P t : \tau$ (where Γ is a type environment of the form $f_1 : \sigma_1, \ldots, f_n : \sigma_n$) means that for any values $v_1, \ldots, v_n$ that have types $\sigma_1, \ldots, \sigma_n$, $[v_1/f_1, \ldots, v_n/f_n]t$ have type τ. We often omit the superscript P when it is clear from the context.

For a program $P = \langle f_1\ \widetilde{x_1} = t_1, \ldots, f_n\ \widetilde{x_n} = t_n \rangle$ and a type τ as a specification, we write $\models P : \tau$ if there exists $\Gamma = f_1 : \sigma_1, \ldots, f_{n-1} : \sigma_{n-1}$ such that $\Gamma \models P(f_n) : \tau$ and $\Gamma \models P(f_i) : \tau_{ij}$ for each $i \in \{1, \ldots, n-1\}$ and $\tau_{ij} \in \sigma_i$. We call such Γ a *witness* for $\models P : \tau$. The goal of our verification is to check whether $\models P : \tau$ holds for a given program P and a refinement type specification τ.

2.3 Examples

In this section, we introduce two examples. Using the first example, we will explain how our method works in Sect. 3.

Example 1. Consider the following program $P_{\texttt{sum}}$:

$$\begin{aligned}\langle\ &\texttt{add}\ x\ y = \mathbf{if}^{\ell_1}\ y \leq 0\ \mathbf{then}\ x\ \mathbf{else}\ 1 + (\texttt{add}\ x\ (y-1)),\\ &\texttt{sum}\ x = \mathbf{if}^{\ell_2}\ x \leq 0\ \mathbf{then}\ 0\ \mathbf{else}\ \texttt{add}\ x\ (\texttt{sum}\ (x-1)),\\ &\texttt{main}\ n = \mathbf{assert}^{\ell_3}(0 \leq \texttt{sum}\ n)\ \rangle.\end{aligned}$$

The function `main` takes an integer n as an argument, computes the sum of integers up to n, and asserts that the sum is no less than 0. The relation $\models P_{\texttt{sum}} : \mathbf{int} \to \mathbf{int}$ means that `main` n never fails for any integer n. It is witnessed by the following type environment $\Gamma_{\texttt{sum}}$:

$$\begin{aligned}&\texttt{add} : \{\{x : \mathbf{int} \mid x \geq 0\} \to \mathbf{int} \to \{r : \mathbf{int} \mid r \geq 0\}\},\\ &\texttt{sum} : \{\mathbf{int} \to \{r : \mathbf{int} \mid r \geq 0\}\},\\ &\texttt{main} : \{\mathbf{int} \to \mathbf{int}\}.\end{aligned}$$

□

Example 2. Consider the following program $P_{\texttt{twice}}$:

$$\begin{aligned}\langle\ &\texttt{mult}\ x\ y = \mathbf{if}^{\ell_1}\ y = 0\ \mathbf{then}\ 0\\ &\qquad\qquad \mathbf{else}\ \mathbf{if}^{\ell_2}\ y < 0\ \mathbf{then}\ -x + \texttt{mult}\ x\ (y+1)\\ &\qquad\qquad \mathbf{else}\ x + \texttt{mult}\ x\ (y-1),\\ &\texttt{twice}\ f\ x = f\ (f\ x),\\ &\texttt{main}\ n = \mathbf{if}^{\ell_3}\ n < 0\ \mathbf{then}\ \mathbf{assert}^{\ell_4}(\texttt{twice}\ (\texttt{mult}\ n)\ 1 > 0)\ \mathbf{else}\ 0\ \rangle.\end{aligned}$$

The function `main` takes an integer n as an argument. If n is negative, then it computes the square of n, and asserts that the square is greater than 0. The relation $\models P_{\texttt{twice}} : \mathbf{int} \to \mathbf{int}$ is witnessed by the following type environment $\Gamma_{\texttt{twice}}$:

$$\begin{aligned}&\texttt{mult} : \{\mathbf{neg} \to \mathbf{neg} \to \mathbf{pos}, \mathbf{neg} \to \mathbf{pos} \to \mathbf{neg}\},\\ &\texttt{twice} : \{\{\mathbf{neg} \to \mathbf{pos}, \mathbf{pos} \to \mathbf{neg}\} \to \mathbf{pos} \to \mathbf{pos}\},\\ &\texttt{main} : \{\mathbf{int} \to \mathbf{int}\}.\end{aligned}$$

Here, **pos** and **neg** are the types of positive and negative integers, which are defined as $\{n : \mathbf{int} \mid n > 0\}$ and $\{n : \mathbf{int} \mid n < 0\}$, respectively. Note here that intersection types are required to make the analysis context-sensitive; in the argument type of `twice`, $\mathbf{neg} \to \mathbf{pos}$ and $\mathbf{pos} \to \mathbf{neg}$ represent the types of the first and second occurrences of f in the body of `twice`, respectively. □

3 An Overview of the Method through an Example

We explain how our method works using the program $P_{\texttt{sum}}$ in Example 1. Suppose we wish to verify that $\models P_{\texttt{sum}} : \mathbf{int} \to \mathbf{int}$.

On lines 2–4 of the overall procedure in Fig. 1, Γ (a type environment that records the types that have been proved valid), $\Gamma_{\texttt{cand}}$ (a candidate type environment), and Π (a set of modular counterexamples) are initialized as follows:

$$\begin{aligned} \Gamma &= \texttt{add} : \emptyset,\ \texttt{sum} : \emptyset,\ \texttt{main} : \emptyset \\ \Gamma_{\texttt{cand}} &= \texttt{add} : \emptyset,\ \texttt{sum} : \emptyset,\ \texttt{main} : \{\mathbf{int} \to \mathbf{int}\} \\ \Pi &= \emptyset \end{aligned}$$

The main procedure then calls `validateTE`, to check the validity of the type of `main`, i.e., whether $\Gamma \models \lambda n.\,\mathbf{assert}(0 \leq \texttt{sum}\ n) : \mathbf{int} \to \mathbf{int}$, by invoking `typeChecker`. To check $\Gamma \models t : \tau$ in general, `typeChecker` uses the technique of Sato et al. [17]: we prepare a context $C_{\Gamma,\tau}$ that is most general in the sense that $C_{\Gamma,\tau}[t]$ fails if and only if $\Gamma \models t : \tau$, and uses a software model checker [9, 18] to check whether $C_{\Gamma,\tau}[t]$ fails. In the case of $\Gamma \models \lambda n.\mathbf{assert}(0 \leq \texttt{sum}\ n) : \mathbf{int} \to \mathbf{int}$, the context $C_{\Gamma,\mathbf{int}\to\mathbf{int}}$ is:

$$\mathbf{let}\ \texttt{add} = \lambda x.\lambda y.\mathbf{fail}\ \mathbf{in}\ \mathbf{let}\ \texttt{sum} = \lambda x.\mathbf{fail}\ \mathbf{in}\ [\,]\ *_{\mathbf{int}}\ .$$

An important point to notice here is that instead of using the original definitions of `add` and `sum`, functions synthesized from their types are used. This enables modular verification of each top-level function. In the present case, since $\Gamma(\texttt{sum})$ is empty, the *weakest* term (in the sense that it is most likely to fail) is chosen as the code of `sum`. A model checker can output the following error path (i.e., a reduction sequence that leads to **fail**):[3]

$$\begin{aligned} & C_{\Gamma,\mathbf{int}\to\mathbf{int}}[\lambda n.\mathbf{assert}(0 \leq \texttt{sum}\ n)] \\ \longrightarrow_P\ & \mathbf{let}\ \texttt{add} = \cdots\ \mathbf{in}\ \mathbf{let}\ \texttt{sum} = \cdots\ \mathbf{in}\ (\lambda n.\mathbf{assert}(0 \leq \texttt{sum}\ n))m \\ \longrightarrow_P\ & \mathbf{let}\ \texttt{add} = \cdots\ \mathbf{in}\ \mathbf{let}\ \texttt{sum} = \cdots\ \mathbf{in}\ \mathbf{assert}(0 \leq \texttt{sum}\ m) \\ \longrightarrow_P\ & \mathbf{let}\ \texttt{add} = \cdots\ \mathbf{in}\ \mathbf{let}\ \texttt{sum} = \cdots\ \mathbf{in}\ \mathbf{assert}(0 \leq (\lambda x.\mathbf{fail})m) \\ \longrightarrow_P\ & \mathbf{let}\ \texttt{add} = \cdots\ \mathbf{in}\ \mathbf{let}\ \texttt{sum} = \cdots\ \mathbf{in}\ \mathbf{assert}(0 \leq \mathbf{fail}) \end{aligned}$$

as a counterexample (where m is some integer). Thus, `typeChecker` can conclude that $\Gamma \models \lambda n.\mathbf{assert}(0 \leq \texttt{sum}\ n) : \mathbf{int} \to \mathbf{int}$ does not hold. The counterexample is useful for refining the candidate type environment, but since it is redundant (it contains information about how the part $C_{\Gamma,\mathbf{int}\to\mathbf{int}}$ is reduced, which is irrelevant to the original program), we keep only information about which branches have been taken inside the term being checked. In the present case, since no branch has been taken, `typeChecker` returns $(\texttt{main}, \epsilon)$ (which means that `main` may fail before encountering any conditional branch) as a *modular* counterexample. Since the only candidate type $\texttt{main} : \mathbf{int} \to \mathbf{int}$ has been rejected, Γ remains to be empty: $\texttt{add} : \emptyset, \texttt{sum} : \emptyset, \texttt{main} : \emptyset$.

[3] Here, for the readability of the reduction sequence, we treat let-expressions as primitives and extend the evaluation contexts with $E ::= \cdots \mid \mathbf{let}\ x = v\ \mathbf{in}\ E$.

Now, `typeSynthesizer` is called to construct a new candidate type environment (line 15), using the modular counterexamples collected so far. The component `typeSynthesizer` prepares a kind of program slice[4] of the original program, which covers all the modular counterexamples. Since $(\texttt{main}, \epsilon)$ is the only counterexample found so far, the following program slice is prepared.

$$\begin{array}{l}\langle\ \texttt{add}\ x\ y = \mathbf{assume}\,(\mathbf{false})\,,\\ \ \ \texttt{sum}\ x = \mathbf{assume}\,(\mathbf{false})\,,\\ \ \ \texttt{main}\ n = \mathbf{let}\ \_ = 0 \leq \texttt{sum}\ n\ \mathbf{in}\ \mathbf{assume}\,(\mathbf{false})\ \rangle.\end{array}$$

The program above contains only the part of the original program that runs the main function up to the first branch; the rest of the code has been replaced by the dummy code $\mathbf{assume}\,(\mathbf{false})$, which just diverges and never fails. We then apply to the above program slice the technique of refinement intersection type inference [21], which is complete for recursion-free programs (modulo a certain assumption on the underlying logic). For the above program, we may obtain the following candidate type environment (note that the type of `sum` has changed):

$$\texttt{add} : \emptyset, \quad \texttt{sum} : \{\mathbf{int} \to \mathbf{int}\}\,, \quad \texttt{main} : \{\mathbf{int} \to \mathbf{int}\}\,.$$

We then recheck whether the new candidate types are valid (line 16). This time, `typeChecker` would fail for `sum`; it tries to prove that $C[\mathbf{fix}(\texttt{sum}, \lambda x.\cdots)]$ does not fail for

$$C \equiv \mathbf{let}\ \texttt{add} = \lambda x.\lambda y.\mathbf{fail}\ \mathbf{in}\ [\,] *_{\mathbf{int}},$$

but finds that the term actually fails when `add` is called. The new modular counterexample $(\texttt{sum}, (\ell_2, \mathbf{else}))$ (which means that the else-branch has been taken at ℓ_2) is then added. Since Γ has not changed, the type checking for `main` also fails again, and `typeSynthesizer` is called with $\Pi = \{(\texttt{sum}, (\ell_2, \mathbf{else})), (\texttt{main}, \epsilon)\}$.

Suppose that the candidate type environment has been further updated, for example, to:

$$\texttt{add} : \{\mathbf{int} \to \mathbf{int} \to \mathbf{int}\}\,, \quad \texttt{sum} : \{\mathbf{int} \to \mathbf{int}\}\,, \quad \texttt{main} : \{\mathbf{int} \to \mathbf{int}\}\,.$$

This time, `typeChecker` succeeds for `add` and `sum`, and $\texttt{add} : \{\mathbf{int} \to \mathbf{int} \to \mathbf{int}\}$, $\texttt{sum} : \{\mathbf{int} \to \mathbf{int}\}$ are added to Γ. The type check for `main` fails, however. To check the type of `main`, `typeChecker` tries to prove that $C'[\lambda n.\mathbf{assert}(0 \leq \texttt{sum}\ n)]$ does not fail for

$$C' = \mathbf{let}\ \texttt{add} = \cdots\ \mathbf{in}\ \mathbf{let}\ \texttt{sum} = \lambda x.*_{\mathbf{int}}\ \mathbf{in}\ [\,] *_{\mathbf{int}},$$

but the term actually fails when $\texttt{sum} = \lambda x.*_{\mathbf{int}}$ returns a negative integer. From the error reduction sequence, the new modular counterexample $(\texttt{main}, (\ell_3, \mathbf{else}))$ is extracted and added to Π. (Recall that $\mathbf{assert}^{\ell}(b)$ is treated as a shorthand form of $\mathbf{if}^{\ell}\ b\ \mathbf{then}\ 1\ \mathbf{else}\ \mathbf{fail}$; thus, the else-branch is taken at ℓ_3 in the error

[4] It is actually an extension of straightline programs [9], and deviates from the standard notion of program slices; see Sect. 4.

reduction sequence.) The component `typeSynthesizer` then discovers that the return type of `sum` should be $\{r : \mathbf{int} \mid r \geq 0\}$.

By repeating these steps, we may end up with the following set of modular counterexamples (we only keep those that are maximal with respect to the prefix relation):

$$\{(\mathtt{add}, (\ell_1, \mathbf{then})),\ (\mathtt{add}, (\ell_1, \mathbf{else})(\ell_1, \mathbf{then})),\\ (\mathtt{sum}, (\ell_2, \mathbf{else})(\ell_2, \mathbf{then})),\ (\mathtt{main}, (\ell_3, \mathbf{else}))\}.$$

The element $(\mathtt{sum}, (\ell_2, \mathbf{else})(\ell_2, \mathbf{then}))$ means that, inside the function `sum`, the else-branch is taken on the first visit of ℓ_2, and then the then-branch is taken on the next visit. The component `typeSynthesizer` constructs the following program slice:

$$\begin{array}{l}\langle\ \mathtt{add}'\ x\ y = \mathbf{if}\ y \leq 0\ \mathbf{then}\ x\ \mathbf{else}\ \mathbf{assume}\,(\mathbf{false}),\\ \quad \mathtt{add}\ x\ y = \mathbf{if}\ y \leq 0\ \mathbf{then}\ x\ \mathbf{else}\ 1 + (\mathtt{add}'\ x\ (y-1)),\\ \quad \mathtt{sum}'\ x = \mathbf{if}\ x \leq 0\ \mathbf{then}\ 0\ \mathbf{else}\ \mathbf{assume}\,(\mathbf{false}),\\ \quad \mathtt{sum}\ x = \mathbf{if}\ x \leq 0\ \mathbf{then}\ \mathbf{assume}\,(\mathbf{false})\ \mathbf{else}\ \mathtt{add}\ x\ (\mathtt{sum}'\ (x-1)),\\ \quad \mathtt{main}\ n = \mathbf{let}\ b = 0 \leq \mathtt{sum}\ n\ \mathbf{in}\ \mathbf{if}\ b\ \mathbf{then}\ \mathbf{assume}\,(\mathbf{false})\ \mathbf{else}\ \mathbf{fail}\ \rangle.\end{array}$$

Here, the functions `add` and `sum` have been duplicated (i) to avoid recursion and (ii) to exclude out the part irrelevant to the modular counterexamples. The refinement (intersection) type inference [23, 21] is applied to the above program slice, and the candidate type environment is updated accordingly to:

$$\begin{array}{l}\mathtt{add} : \{\{x : \mathbf{int} \mid x \geq 0\} \to \mathbf{int} \to \{r : \mathbf{int} \mid r \geq 0\}\},\\ \mathtt{sum} : \{\mathbf{int} \to \{r : \mathbf{int} \mid r \geq 0\}\},\quad \mathtt{main} : \{\mathbf{int} \to \mathbf{int}\}.\end{array}$$

The component `typeChecker` can now successfully verify that all the above types are valid, and add them to Γ. Since Γ now contains $\mathbf{int} \to \mathbf{int}$ as a type of `main`, the verification succeeds (on line 13 of Fig. 1). The final type environment that has been proved valid is:

$$\begin{array}{l}\mathtt{add} : \{\mathbf{int} \to \mathbf{int} \to \mathbf{int},\ \{x : \mathbf{int} \mid x \geq 0\} \to \mathbf{int} \to \{r : \mathbf{int} \mid r \geq 0\}\},\\ \mathtt{sum} : \{\mathbf{int} \to \mathbf{int},\ \mathbf{int} \to \{r : \mathbf{int} \mid r \geq 0\}\},\\ \mathtt{main} : \{\mathbf{int} \to \mathbf{int}\}.\end{array}$$

The example above is oversimplified in that neither higher-order functions nor local function definitions occur, and that intersection types are not used. We present our method more formally in the next section.

4 Verification Method

This section describes our verification method in detail. As mentioned in Sect. 1, our method consists of the two components `typeChecker` and `typeSynthesizer`, which are described in Sects. 4.1 and 4.2, respectively.

4.1 `typeChecker`: Checking type candidate

The method `typeChecker` verifies whether

$$f_1 : \sigma_1, \ldots, f_{k-1} : \sigma_{k-1} \models \mathbf{fix}(f_k, \lambda \widetilde{x}_k.\, t_k) : \tau_k$$

holds for each $k \in \{1, \ldots, n\}$, given the program $\langle f_1\, \widetilde{x}_1 = t_1, \ldots, f_n\, \widetilde{x}_n = t_n \rangle$, the current type environment $f_1 : \sigma_1, \ldots, f_n : \sigma_n$, and the current refinement type candidate τ_n of f_n.

We reduce a type judgment $\Gamma \overset{?}{\models} t : \tau$ to a safety checking problem by using an extension of Sato et al.'s method [17]. For example, the type checking problem

$$f : (\{x : \mathbf{int} \mid x > 0\} \to \{r : \mathbf{int} \mid r \geq x\}) \overset{?}{\models} t : \{y : \mathbf{int} \mid y \neq 0\} \to \{s : \mathbf{int} \mid s > y\}$$

is reduced to the safety checking problem for the following program:

$$\begin{array}{l} \mathbf{let} f = \lambda x.\, \mathbf{if}\ x > 0\ \mathbf{then\ let}\ r = *_{\mathbf{int}}\ \mathbf{in\ assume}\,(r \geq x);\, r \\ \qquad\qquad \mathbf{else\ fail\ in} \\ \mathbf{let}\ y = \mathbf{let}\ y' = *_{\mathbf{int}}\ \mathbf{in\ assume}\,(y' \neq 0);\, y'\ \mathbf{in} \\ \mathbf{let}\ s = t\, y\ \mathbf{in\ assert}(s > y) \end{array}$$

Here, the bodies of f and y are "universal" terms of types $\{x : \mathbf{int} \mid x > 0\} \to \{r : \mathbf{int} \mid r \geq x\}$ and $\{y : \mathbf{int} \mid y \neq 0\}$, respectively. A universal term t of type τ can simulate all the values of type τ, in the sense that, for any context C, term t' of type τ, and integer n, $C[t'] \longrightarrow^* n$ implies $C[t] \longrightarrow^* n$, and $C[t'] \longrightarrow^* \mathbf{fail}$ implies $C[t] \longrightarrow^* \mathbf{fail}$.

As seen above, by using universal terms, we can reduce a type judgment problem to a safety problem. In general, there exists a most general context $C_{\Gamma,\tau}$ with respect to a type environment Γ and a refinement type τ such that

$$C_{\Gamma,\tau}[t] \not\longrightarrow^* \mathbf{fail} \quad \text{if and only if} \quad \Gamma \models t : \tau$$

for any t such that $ST(\Gamma) \vdash_{\mathrm{ST}} t : ST(\tau)$. Hence, we can check $\Gamma \models t : \tau$ by checking the safety of term $C_{\Gamma,\tau}[t]$. If the term is safe, then t has type τ, and otherwise, t does not have type τ for some $f_1, \ldots, f_n$ that have types $\Gamma(f_1), \ldots, \Gamma(f_n)$, respectively.

Note that, even if the term $C_{\Gamma,\tau}[t]$ is unsafe, we cannot conclude that t does not have type τ in the original program P. The unsafety of $C_{\Gamma,\tau}[t]$ just indicates the untypability of t under the given type environment Γ, i.e., the type environment is too weak to prove the typability of t.

Sato et al. [17] formalized the construction of $C_{\Gamma,\tau}$ for refinement types *without* intersection types. Below we extend their method to deal with refinement intersection types. We define the most general context $C_{\Gamma,\tau}$ with respect to Γ and τ by using universal terms. The universal term synthesizer $\alpha_\wedge(-)$ is defined in Fig. 5. The function $\alpha_\wedge(\sigma)$ ($\alpha(\tau)$, resp.) synthesizes a universal term of

$$\alpha\left(\{x : \mathbf{int} \mid P\}\right) = \mathbf{let}\ x = *_{\mathbf{int}}\ \mathbf{in}\ \mathbf{assume}\left(P\right);\ x$$
$$\alpha\left((x : \sigma) \to \tau\right) = \lambda x.\, \mathbf{if}\ *_{\mathbf{bool}} \vee \beta_{\wedge}\left(x : \sigma\right)\ \mathbf{then}\ \alpha\left(\tau\right)\ \mathbf{else}\ \mathbf{fail}$$
$$\alpha_{\wedge}\left(\left\{\left\{x : \mathbf{int} \mid P_1\right\}, \ldots, \left\{x : \mathbf{int} \mid P_n\right\}\right\}_{\mathbf{int}}\right) = \alpha\left(\left\{x : \mathbf{int} \mid P_1 \wedge \cdots \wedge P_n\right\}\right)$$
$$\alpha_{\wedge}\left(\left\{(x : \sigma_1) \to \tau_1, \ldots, (x : \sigma_n) \to \tau_n\right\}_{\kappa_1 \to \kappa_2}\right) =$$
$$\quad \mathit{wrap}((x : \sigma_1) \to \tau_1, \mathit{wrap}(\cdots, \mathit{wrap}((x : \sigma_n) \to \tau_n, \lambda x.\, \mathbf{fail}) \cdots))$$
where $\mathit{wrap}((x : \sigma) \to \tau, v) =$
$$\quad \lambda x.\, \mathbf{let}\ f = v\ \mathbf{in}$$
$$\quad\quad \mathbf{if}\ *_{\mathbf{bool}} \vee \beta_{\wedge}\left(x : \sigma\right)\ \mathbf{then}$$
$$\quad\quad\quad \mathbf{try}\ \mathbf{let}\ r = f\, x\ \mathbf{in}\ \mathit{wrap}(\tau, r)\ \mathbf{with}\ \mathbf{fail} \to \alpha\left(\tau\right)$$
$$\quad\quad \mathbf{else}\ f\, x$$
$$\quad \mathit{wrap}(\{x : \mathbf{int} \mid P\}, v) = \mathbf{assume}\left([v/x]P\right);\ v$$

$$\beta\left(v : \{x : \mathbf{int} \mid P\}\right) = [v/x]P$$
$$\beta\left(v : (x : \sigma) \to \tau\right) = \mathbf{let}\ x = \alpha_{\wedge}\left(\sigma\right)\ \mathbf{in}\ \mathbf{let}\ r = v\, x\ \mathbf{in}\ \beta\left(r : \tau\right)$$
$$\beta_{\wedge}\left(v : \{\tau_1, \ldots, \tau_n\}_{\kappa}\right) = \beta\left(v : \tau_1\right) \wedge \cdots \wedge \beta\left(v : \tau_n\right)$$

Fig. 5. Synthesis of universal terms

type σ (τ, resp.). The function $\beta_{\wedge}\left(v : \sigma\right)$ ($\beta\left(v : \tau\right)$, resp.) checks whether v has type σ (τ, resp.). If $\beta_{\wedge}\left(v : \sigma\right)$ returns **false** or aborts with **fail**, then v does not have type σ. In the case of an intersection of function types, we use exceptions and treat **fail** as an exception, which can be removed by CPS transformation. The function $\mathit{wrap}(\tau, v)$, intuitively, forces v to have type τ by inserting assume expressions into v. For integer types, $\mathit{wrap}(\{x : \mathbf{int} \mid P\}, v)$ just assumes $[v/x]P$ and returns v. For functions types, $\mathit{wrap}((x : \sigma) \to \tau, v)$ returns a new function that is an eta-expansion of v and in which assume expressions are inserted. In the body of the new function, if the then-branch is taken (which indicates that the argument x may have type σ), the return value must have type τ. If $f\, x$ is evaluated to some value r, then the new function returns $\mathit{wrap}(\tau, r)$, which is forced to have type τ. If the evaluation of $f\, x$ fails, then the new function returns the universal term of τ. If the else-branch is taken (which indicates that the argument x does not have type σ), since the return value of the new function need not have type τ, $\mathit{wrap}((x : \sigma) \to \tau, v)$ returns the original result $f\, x$. Note that we can remove "$*_{\mathbf{bool}} \vee$ " in the definitions of $\alpha\left((x : \sigma) \to \tau\right)$ and $\mathit{wrap}((x : \sigma) \to \tau, v)$, if we know that $\beta_{\wedge}\left(x : \sigma\right)$ terminates. Especially, we can remove "$*_{\mathbf{bool}} \vee$ " when σ is an integer type like the example above.

By using $\alpha_\wedge(-)$ and $\beta(-:-)$, the most general context $C_{\Gamma,\tau}$ can be defined as follows:

$$C_{(f_1:\sigma_1,\ldots,f_n:\sigma_n),\tau} = \textbf{let } f_1 = \alpha_\wedge(\sigma_1) \textbf{ in } \ldots \textbf{ let } f_n = \alpha_\wedge(\sigma_n) \textbf{ in} \\ \textbf{let } f = [\,] \textbf{ in assert}(\beta(f:\tau))$$

The following lemma states the correctness of the construction of $C_{\Gamma,\tau}$, which can be proved in a manner similar to the original construction of Sato et al. [17] for refinement types without intersections.

Lemma 1. *Suppose* $ST(\Gamma) \vdash_{\mathrm{ST}} t : ST(\tau)$.

$$\Gamma \models^P t:\tau \quad \textit{if and only if} \quad C_{\Gamma,\tau}[t] \not\longrightarrow_P^* \textbf{fail}.$$

The reduced problem can be checked by an existing safety checker (e.g., MoCHi [9, 18]) that satisfies the following properties:

- It can check the safety of a given program t, i.e., whether $t \not\longrightarrow_P^* \textbf{fail}$.
- It can generate a counterexample, i.e., a concrete reduction sequence of the form $t \longrightarrow_P^* \textbf{fail}$, given an unsafe program.

We use counterexamples obtained by the checker to find type candidates of top-level functions. Instead of using the counterexamples themselves, we use their subsequence related to the target function. We call them *modular counterexamples*. A modular counterexample π of top-level function f is a sequence of pairs of labels and branching information $\{\textbf{then}, \textbf{else}\}$, i.e., $\pi : (L \times \{\textbf{then}, \textbf{else}\})^*$ where L is the set of labels.

A modular counterexample of f is obtained from an ordinary counterexample π as follows. We write $L(t)$ for the set of the labels occurred in t, and write L_f for $L(t)$ where $(f\,\widetilde{x} = t) \in P$. Suppose the given counterexample π is of the following form

$$\begin{aligned} C_{\Gamma,\tau}[t] &\longrightarrow^* E_1[\textbf{if}^{\ell_1}\ v_1 \textbf{ then } t_{12} \textbf{ else } t_{13}] \\ &\longrightarrow^* E_2[\textbf{if}^{\ell_2}\ v_2 \textbf{ then } t_{22} \textbf{ else } t_{23}] \\ &\quad\vdots \\ &\longrightarrow^* E_n[\textbf{if}^{\ell_n}\ v_n \textbf{ then } t_{n2} \textbf{ else } t_{n3}] \\ &\longrightarrow^* \textbf{fail}. \end{aligned}$$

Then, a modular counterexample of function f is

$$(\ell_{j_1}, b_{j_1}) \ldots (\ell_{j_k}, b_{j_k}) \quad \text{where } 1 \le j_1 < \cdots < j_k \le n, \\ \{j_1, \ldots, j_k\} = \{j \mid \ell_j \in L_f\}, \text{ and} \\ b_j = \begin{cases} \textbf{then} & v_j \neq 0 \\ \textbf{else} & v_j = 0 \end{cases} \quad \text{for each } j \in \{j_1, \ldots, j_k\}.$$

Example 3. Recall the program $P_{\texttt{sum}}$ in Example 1. Suppose that $\tau = \mathbf{int} \to \{r : \mathbf{int} \mid r = 0\}$ is given as a type candidate of `sum` and the following type environment is given:

$$\Gamma = \texttt{add} : \{\mathbf{int} \to \mathbf{int} \to \mathbf{int}\}\,.$$

Then, the most general context $C_{\Gamma,\tau}$ is

$$\begin{aligned} C_{\Gamma,\tau} = {} & \mathbf{let}\ \texttt{add} = \lambda x.\, \lambda y.\, *_{\mathbf{int}}\ \mathbf{in} \\ & \mathbf{let}\ x = *_{\mathbf{int}}\ \mathbf{in}\ \mathbf{let}\ r = [\,]\, x\ \mathbf{in}\ \mathbf{assert}(r = 0). \end{aligned}$$

Since `sum` does not have type τ under the type environment, we have the following counterexample for some $m \neq 0$:

$$\begin{aligned} & C_{\Gamma,\tau}[t_{\texttt{sum}}] \\ \longrightarrow^* {} & \mathbf{let}\ r = t'_{\texttt{sum}}\, 1\ \mathbf{in}\ \mathbf{assert}(r = 0) \\ \longrightarrow^* {} & \mathbf{let}\ r = \mathbf{if}^{\ell_2}\ 1 \leq 0\ \mathbf{then}\ 0\ \mathbf{else}\ t_{\texttt{add}}\, 1\, (t'_{\texttt{sum}}\, (1-1))\ \mathbf{in}\ \mathbf{assert}(r = 0) \\ \longrightarrow^* {} & \mathbf{let}\ r = t_{\texttt{add}}\, 1\, (\mathbf{if}^{\ell_2}\ 0 \leq 0\ \mathbf{then}\ 0\ \mathbf{else}\ \ldots)\ \mathbf{in}\ \mathbf{assert}(r = 0) \\ \longrightarrow^* {} & \mathbf{let}\ r = t_{\texttt{add}}\, 1\, 0\ \mathbf{in}\ \mathbf{assert}(r = 0) \\ \longrightarrow^* {} & \mathbf{let}\ r = m\ \mathbf{in}\ \mathbf{assert}(r = 0) \\ \longrightarrow^* {} & \mathbf{fail} \end{aligned}$$

where

$$\begin{aligned} & t_{\texttt{sum}} = P(\texttt{sum}) = \mathbf{fix}(\texttt{sum}, \lambda x.\, \mathbf{if}^{\ell_2}\ x \leq 0\ \mathbf{then}\ 0\ \mathbf{else}\ \texttt{add}\ x\ (\texttt{sum}\ (x-1))) \\ & t'_{\texttt{sum}} = [t_{\texttt{add}}/\texttt{add}]t_{\texttt{sum}} \\ & t_{\texttt{add}} = \lambda x.\, \lambda y.\ *_{\mathbf{int}}\ . \end{aligned}$$

There are two branches labeled with ℓ_2, which occurs in the body of `sum`. The else-branch is taken on the first visit of ℓ_2, and the then-branch is taken on the next visit. We then obtain the following modular counterexample:

$$(\texttt{sum}, (\ell_2, \mathbf{else})(\ell_2, \mathbf{then})).$$

□

4.2 `typeSynthesizer`: Synthesizing new refinement types

The function `typeSynthesizer` finds type candidates by using the modular counterexamples found so far. It first generates a program slice of the original program corresponding to modular counterexamples, and infers a refinement type of the program slice. The inferred refinement type can be used as a type candidate of the original program.

Given a set of modular counterexamples

$$\Pi \subseteq \mathcal{P}\left(dom(P) \times (L \times \{\mathbf{then}, \mathbf{else}\})^*\right),$$

we generate a program slice of $P(f_i)$ that corresponds to Π, for which we write D_{P,Π,f_i}. We first construct a computation tree whose path corresponds to a execution trace that follows the modular counterexamples. The corresponding program D_{P,Π,f_i} is obtained by (i) making a copy of each function for each call in the computation tree, and (ii) for each copy, removing the branches not taken in the corresponding execution trace.

Example 4. Recall the program $P_{\texttt{sum}}$ in Example 3. Suppose the target function and the type are `main` and $\mathbf{int} \to \mathbf{int}$, and the following set Π of modular counterexamples is given:

$$\begin{aligned}\{\ &(\texttt{add}, (\ell_1, \mathbf{then})),\\ &(\texttt{add}, (\ell_1, \mathbf{else})(\ell_1, \mathbf{then})),\\ &(\texttt{sum}, (\ell_2, \mathbf{else})(\ell_2, \mathbf{else})(\ell_2, \mathbf{then})),\\ &(\texttt{main}, (\ell_3, \mathbf{else}))\ \}.\end{aligned}$$

Then the program $D_{P_{\texttt{sum}},\Pi,\texttt{main}}$ corresponding to the modular counterexamples is

$$\begin{aligned}\langle\ &\texttt{add}_1\ x\ y = \mathbf{if}\ y \leq 0\ \mathbf{then}\ x\ \mathbf{else}\ \mathbf{assume}\,(\mathbf{false}),\\ &\texttt{add}'_2\ x\ y = \mathbf{if}\ y \leq 0\ \mathbf{then}\ x\ \mathbf{else}\ \mathbf{assume}\,(\mathbf{false}),\\ &\texttt{add}_2\ x\ y = \mathbf{if}\ y \leq 0\ \mathbf{then}\ \mathbf{assume}\,(\mathbf{false})\ \mathbf{else}\ 1 + \texttt{add}'_2\ x\ (y-1),\\ &\texttt{add}\ x\ y = \texttt{add}_1\ x\ y \,\square\, \texttt{add}_2\ x\ y,\\ &\texttt{sum}''_1\ x = \mathbf{if}\ x \leq 0\ \mathbf{then}\ 0\ \mathbf{else}\ \mathbf{assume}\,(\mathbf{false}),\\ &\texttt{sum}'_1\ x = \mathbf{if}\ x \leq 0\ \mathbf{then}\ \mathbf{assume}\,(\mathbf{false})\ \mathbf{else}\ \texttt{add}\ x\ (\texttt{sum}''_1\ (x-1)),\\ &\texttt{sum}_1\ x = \mathbf{if}\ x \leq 0\ \mathbf{then}\ \mathbf{assume}\,(\mathbf{false})\ \mathbf{else}\ \texttt{add}\ x\ (\texttt{sum}'_1\ (x-1)),\\ &\texttt{sum}\ x = \texttt{sum}_1\ x,\\ &\texttt{main}_1\ n = \mathbf{if}\ 0 \leq \texttt{sum}\ n\ 0\ \mathbf{then}\ \mathbf{assume}\,(\mathbf{false})\ \mathbf{else}\ \mathbf{fail},\\ &\texttt{main}\ n = \texttt{main}_1\ n\ \rangle.\end{aligned}$$

A function corresponding to each modular counterexample is generated: $\texttt{add}_1$ from $(\texttt{add}, (\ell_1, \mathbf{then}))$, $\texttt{add}_2$ from $(\texttt{add}, (\ell_1, \mathbf{else})(\ell_1, \mathbf{then}))$, $\texttt{sum}_1$ from $(\texttt{sum}, (\ell_2, \mathbf{else})(\ell_2, \mathbf{else})(\ell_2, \mathbf{then}))$, and $\texttt{main}_1$ from $(\texttt{main}, (\ell_3, \mathbf{else}))$. The function `typeSynthesizer` then infers a refinement type of $C_{\emptyset,\mathbf{int}\to\mathbf{int}}[D_{P_{\texttt{sum}},\Pi,\texttt{main}}]$, and obtains the following types:

$$\begin{aligned}&\texttt{add} : \{\{x : \mathbf{int} \mid x \geq 0\} \to \mathbf{int} \to \{r : \mathbf{int} \mid r \geq 0\}\},\\ &\texttt{sum} : \{\mathbf{int} \to \{r : \mathbf{int} \mid r \geq 0\}\}.\end{aligned}$$

We use the above types as type candidates of `add` and `sum`. □

If the constructed program is not typable, so is the original program. Then, the function `typeSynthesizer` answers "There are no candidates" and our method returns "no". In this case, we can obtain an untypable execution trace, and output the trace as an ordinary counterexample.

The concrete definition of `typeSynthesizer` is shown in Appendix A. The construction is similar to that of straightline programs used in MoCHi [9].

The following lemma guarantees that the modular counterexample π is indeed a counterexample in that a slice of $P(f_i)$ containing a path corresponding to π is indeed (semantically) untypable.

Lemma 2. *Let P be a program, and π be a modular counterexample against $\Gamma \models P(f_i) : \tau_i$. If $\pi \in \Pi$ and D_{P,Π,f_i} is the slice of $P(f_i)$ corresponding to Π, then $\Gamma \not\models D_{P,\Pi,f_i} : \tau_i$.*

4.3 Properties of the Method

We now discuss properties of our method. The method is sound (under the assumption that the underlying verifier is sound), in the sense that, if the method returns "yes" ("no", resp.), then the given program has (does not have, resp.) the given type. This is an easy consequence of the soundness of `typeChecker`, i.e., the soundness of the reduction from refinement type checking to assertion checking.

Our method also satisfies a progress property, in that the set of modular counterexamples monotonically increases until the method terminates. More precisely, in the overall procedure in Fig. 1, either Γ or Π strictly increases upon each recursive call of `validateTE`. We can prove the progress as follows. Suppose that `validateTE` is called with a non-empty candidate type environment $\Gamma_{\mathtt{cand}}$, that Γ does not change in the for-loop, and that $\tau \in \Gamma(f_n)$ does not hold on line 13. Let i be the least i such that $\Gamma_{\mathtt{cand}}(f_i) \neq \emptyset$, and there exists $\tau' \in \Gamma_{\mathtt{cand}}(f_i)$ such that $\Gamma \not\models P(f_i) : \tau'$; note that there always exists such i by the assumption that $\tau \in \Gamma(f_n)$ does not hold on line 13. Since $\Gamma_{\mathtt{cand}}(f_i) \neq \emptyset$, $\mathtt{typeChecker}(\Gamma, P(f_i), \tau')$ returns $\mathtt{NG}(\pi)$ for some π. We show $\pi \notin \Pi$ by contradiction. Suppose $\pi \in \Pi$. By Lemma 2, $\Gamma \not\models D_{P,\Pi,f_i} : \tau'$, where D_{P,Π,f_i} is the slice of $P(f_i)$ corresponding to Π. This contradicts $\tau' \in \Gamma_{\mathtt{cand}}(f_i)$, since in the previous call of `validateTE`, $\Gamma_{\mathtt{cand}}$ has been constructed from Π (so, τ' has been chosen so that $\Gamma \models D_{P,\Pi,f_i} : \tau'$ holds). Thus, we have $\pi \notin \Pi$, which implies that Π strictly increases on line 12.

With a certain assumption on the underlying reachability checker used in `typeChecker`, we can also guarantee the completeness for finding a counterexample. Suppose that, for the problem $C_{\Gamma,\tau}[P(f_i)] \stackrel{?}{\longrightarrow}_P^* \mathbf{fail}$ obtained from a type checking problem $\Gamma \stackrel{?}{\models} P(f_i) : \tau$, if there is a counterexample, the reachability checker returns the one corresponding to the *least* (with respect to a certain total order on modular counterexamples) modular counterexample that does not belong to Π (if there is any). Then, by the progress property, every counterexample is eventually enumerated, so that a counterexample to the original verification problem is eventually found if there is any.

In order to guarantee the relative completeness for verification in the sense of [24] (i.e., if $\models P : \tau$, then the method is eventually able to prove it, modulo a certain assumption on the underlying logic), we need to extend the method

program	LOC	#module	MoCHi [sec]	modular [sec]	#`typeChecker`
sum_add	3	3	0.57	1.64	11
harmonic	18	4	0.88	5.48	17
fold_div	19	4	0.86	5.71	18
risers	21	3	8.93	2.66	4
various	23	3	TIMEOUT	0.04	5
colwheel	69	5	TIMEOUT	25.06	7
queen	45	4	5.69	9.70	8
queen_simple	20	2	TIMEOUT	14.86	7
soli	93	5	TIMEOUT	17.84	8
spir	75	11	5.06	48.94	21
doctor	568	12	TIMEOUT	543.93	45
various-e	23	3	0.34	2.24	5
queen_simple-e	19	2	0.74	4.02	5

Table 1. Results of experiments

to automatically infer implicit parameters (as in [24]) for each function module, which is left for future work.

5 Experiments

We have implemented an automated verification tool for a subset of OCaml, based on the proposed method. We use MoCHi [9] as the backend safety checker used in `typeChecker`. We have tested our tool for programs taken from the benchmark for MoCHi and Caml Examples [27]. We have conducted the experiments on a machine with Intel Core i7-3930K (3.20 GHz, 16 GB of memory), with timeout of 600 seconds. All the programs are available on the web `http://www-kb.is.s.u-tokyo.ac.jp/~ryosuke/modular/`.

Table 1 summarizes the experimental results. The column "program" shows the names of the programs. The column "LOC" shows the number of lines of code excluding comments and blank lines. The column "#module" shows the number of modules, i.e., top-level functions. The columns "MoCHi" and "modular" show the running time in seconds of the original MoCHi and our new verifier respectively. The column "#`typeChecker`" shows the number of calls to `typeChecker`.

All the benchmark programs are safe except `various-e` and `queen_simple-e`, i.e., they are free from assertion failures, pattern matching failures, uncaught exceptions, and array bound errors. We explain each benchmark program below. The program `sum_add` is $P_{\texttt{sum}}$ in Example 1. The programs `harmonic`, `fold_div`, and `risers` have been taken from the benchmark of MoCHi [18]. We have chosen the programs which are no less than 18 lines and have no less than 3 modules. The program `various` is a composition of small programs taken from the benchmark of MoCHi, namely `sum`, `mult`, and `mc91`.

The other programs `colwheel`–`doctor` have been taken from Caml Examples [27]. The program `colwheel` displays a color chart, which uses exceptions and variants defined in Graphics module. The program `queen` solves the eight queen problem, which uses arrays. We encode arrays as functions, and insert assertions on array bounds. We insert assertions that the index used in an operation on array is no less than 0. The program `queen_simple` is a simplified version of `queen`, but the assertions on array bounds are more strict than `queen`. We also insert assertions that the index used in an operation is less than the size of the array. The program `soli` solves a Peg solitaire game, which also uses exceptions, variants, and arrays. The program `spir` shows an animation of a colorful spiral, which uses an array. The program `doctor` is a chatterbot, which uses exceptions. A program of name "xxx-e" is a buggy version of the program "xxx".

As seen in Table 1, our new tool successfully verifies all the benchmark programs, whereas MoCHi failed to verify `various`, `colwheel`, `queen_simple`, `soli`, and `doctor` in 600 seconds. For the other programs (that MoCHi could also verify) except `risers`, our new tool is actually slower than MoCHi. For those programs, `typeChecker` was called many times before appropriate refinement types were discovered. There is an obvious trade-off between the modular and whole program verification; in reasoning about each function, the latter can use more precise information about the other functions. We expect that the advantage of the modular verification is clearer for larger programs.

6 Related Work

As mentioned in Sect. 1, most of the fully-automated verification methods for higher-order functional programs [9, 15, 18, 10, 13, 14, 26] are whole program analyses. The exceptions are those based on refinement type inference [21, 30, 31], which have similarities to our method in that they consist of two components: one to infer candidate refinement types of functions, and the other to check the validity of the candidate refinement types; the latter can be carried out in a compositional manner, based on a refinement type system. Each component is, however, significantly different from ours. For the first component, Terauchi [21] applies the technique of refinement type inference [23] to the recursion-free programs obtained by finitely unfolding recursive functions, whereas Zhu et al. [30] apply a machine learning technique. Our `typeSynthesizer` component is closer to Terauchi's one [21], but only looks at a part of the program relevant to modular counterexamples found so far. It would be interesting to integrate Zhu et al.'s machine learning technique [30, 31] into our `typeSynthesizer` component, which is left for future work. For the second component, both Terauchi [21] and Zhu et al. [30] use a specific set of syntactic typing rules for refinement types, which is not complete with respect to the semantic refinement type judgment. Our `typeChecker` component reduces the semantic type judgment to a reachability checking problem and delegates the latter to a software model checker [9, 24], so that the component is relatively complete in the sense of [24]. As a re-

sult, our modular verification tool is as powerful as MoCHi, and can generate a concrete error path as a counterexample if a given program does not satisfy a specification, unlike Terauchi and Zhu et al.'s methods [21, 30]. For the `typeChecker` component, we have extended Sato et al.'s technique [17] to deal with intersection types. Voirol et al. [26] and Unno et al. [23] reduce the verification of higher-order programs to the satisfiability checking of quantifier-free formulas and Horn clauses, respectively, and then use constraint solvers; thus, the scalability of the methods depends on those of the underlying solvers. We are not aware of a good modular method for checking the satisfiability.

In contrast with fully-automated verification methods, semi-automated verification methods for functional programs [28, 16, 29, 11, 6] usually work in a compositional manner. Those methods, however, rely on annotations of invariants (or predicates used in invariants [16]). Among them, liquid types [16, 25] require less annotations. Since the liquid types also rely on syntactic refinement typing rules, the comment above on Zhu et al. and Terauchi's methods [21, 30] applies.

For finite state systems, a lot of techniques have been proposed for compositional verification [22, 2, 8, 3, 1, 5, 7, 4]. Some of them infer the interfaces of components based on lazy parallel composition [22, 3] and assume-guarantee reasoning [5, 7]. It is not clear how to extend those methods to deal with higher-order functional programs.

7 Conclusion

We have proposed an automated modular verification method for higher-order functional programs. We have introduced the notion of modular counterexamples to infer candidate refinement intersection types of each function module, and extended Sato et al.'s method [17] to check the validity of the inferred candidate types in a modular manner. We have implemented the proposed method and confirmed its effectiveness through experiments.

Further optimizations are required to make our verification tool more scalable for larger programs. Future work also includes a relatively complete modular verification method (recall the discussion at the end of Sect. 4.3), and extensions of the modular method for proving liveness properties.

Acknowledgment

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

References

1. Berezin, S., Campos, S., Clarke, E.M.: Compositional reasoning in model checking. In: Proceedings of Revised Lectures from the International Symposium on Compositionality: The Significant Difference (COMPOS '97). pp. 81–102 (1998)

2. Burch, J., Clarke, E.M., Long, D.: Symbolic model checking with partitioned transition relations. In: Proceedings of the IFIP TC10/WG 10.5 International Conference on Very Large Scale Integration (VLSI 1991). pp. 49–58 (1991)
3. Campos, S.V.A.: A quantitative approach to the formal verification of real-time systems. Ph.D. thesis, Carnegie Mellon University (1996)
4. Chaki, S., Gurfinkel, A.: Automated assume-guarantee reasoning for omega-regular systems and specifications. Innovations in Systems and Software Engineering 7(2), 131–139 (2011)
5. Cobleigh, J.M., Giannakopoulou, D., Psreanu, C.S.: Learning assumptions for compositional verification. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003). pp. 331–346 (2003)
6. Filliâtre, J.C., Paskevich, A.: Why3 - where programs meet provers. In: Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP 2013). pp. 125–128 (2013)
7. Gheorghiu Bobaru, M., Psreanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Proceedings of the 20th International Conference on Computer Aided Verification (CAV 2008). pp. 135–148 (2008)
8. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems 16(3), 843–871 (1994)
9. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011). pp. 222–233 (2011)
10. Kobayashi, N., Tabuchi, N., Unno, H.: Higher-order multi-parameter tree transducers and recursion schemes for program verification. In: Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010). pp. 495–508 (2010)
11. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2010). pp. 348–370 (2010)
12. Matsumoto, Y., Kobayashi, N., Unno, H.: Automata-based abstraction for automated verification of higher-order tree-processing programs. In: Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS 2015). pp. 295–312 (2015)
13. Nguyen, P.C., Horn, D.V.: Relatively complete counterexamples for higher-order programs. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015). pp. 446–456. ACM (2015)
14. Nguyen, P.C., Tobin-Hochstadt, S., Horn, D.V.: Soft contract verification. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP 2014). pp. 139–152 (2014)
15. Ong, C.H.L., Ramsay, S.J.: Verifying higher-order functional programs with pattern-matching algebraic data types. In: Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011). pp. 587–598 (2011)
16. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008). pp. 159–169 (2008)

17. Sato, R., Asada, K., Kobayashi, N.: Refinement type checking via assertion checking. Journal of Information Processing 23(6), 827–834 (2015)
18. Sato, R., Unno, H., Kobayashi, N.: Towards a scalable software model checker for higher-order programs. In: Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM 2013). pp. 53–62. ACM Press (2013)
19. Swamy, N., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S., Hricu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Swamy, N., Hricu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016). pp. 256–270 (2016)
20. Terao, T., Tsukada, T., Kobayashi, N.: Higher-order model checking in direct style. In: Proceedings of the 14th Asian Symposium on Programming Languages and Systems (APLAS 2016). pp. 295–313 (2016)
21. Terauchi, T.: Dependent types from counterexamples. In: Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010). pp. 119–130 (2010)
22. Touati, H., Savoj, H., Lin, B., Brayton, R., Sangiovanni-Vincentelli, A.: Implicit state enumeration of finite state machines using BDD's. In: 1990 IEEE International Conference on Computer-Aided Design (ICCAD-90). pp. 130–133 (1990)
23. Unno, H., Kobayashi, N.: Dependent type inference with interpolants. In: Proceedings of the 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2009). pp. 277–288 (2009)
24. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2013). pp. 75–86 (2013)
25. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Jones, S.L.P.: Refinement types for haskell. In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming. pp. 269–282 (2014)
26. Voirol, N., Kneuss, E., Kuncak, V.: Counter-example complete verification for higher-order functions. In: Proceedings of the 6th ACM SIGPLAN Symposium on Scala (Scala 2015). pp. 18–29 (2015)
27. Weis, P.: Caml examples (2001), `http://caml.inria.fr/pub/old_caml_site/Examples/`
28. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1999). pp. 214–227 (1999)
29. Zhu, H., Jagannathan, S.: Compositional and lightweight dependent type inference for ML. In: Proceedings of the 14th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2013). pp. 295–314 (2013)
30. Zhu, H., Nori, A.V., Jagannathan, S.: Learning refinement types. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015). pp. 400–411 (2015)
31. Zhu, H., Petri, G., Jagannathan, S.: Automatically learning shape specifications. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI 2016). pp. 491–507 (2016)

$$\frac{f \notin F_{\mathtt{top}} \qquad (f\,\widetilde{x} = e_1 \,\Box^\ell\, e_2) \in D}{(D, f\,\widetilde{v}) \longrightarrow (D, [\widetilde{v}/\widetilde{x}](e_1 \,\Box^\ell\, e_2))}$$

$$\big(D, e_1 \,\Box^\ell\, e_2\big) \longrightarrow (D, e_b)$$

$$(D, \mathbf{assume}\,(\mathbf{true})\,; e) \longrightarrow (D, e)$$

$$(D, \mathbf{let}\ x = *_{\mathbf{int}}\ \mathbf{in}\ a) \longrightarrow (D, [n/x]a)$$

Fig. 6. Operational semantics of the intermediate language

A Definition of `typeSynthesizer`

This section gives the definition of the component `typeSynthesizer`. For the simplicity, we use the intermediate language defined as follows:

$$\begin{aligned} D &::= \big\{ f_1\,\widetilde{x}_1 = e_{10} \,\Box^\ell\, e_{11}, \ldots, f_m\,\widetilde{x}_m = e_{m0} \,\Box\, e_{m1} \big\} \\ e &::= a \mid \mathbf{assume}\,(v)\,;\, e \mid \mathbf{let}\ x = *_{\mathbf{int}}\ \mathbf{in}\ e \\ a &::= \langle\rangle \mid x\,\widetilde{v} \mid f\,\widetilde{v} \mid \mathbf{fail} \\ v &::= c \mid x\,\widetilde{v} \mid f\,\widetilde{v} \mid \mathtt{op}(\widetilde{v}). \end{aligned}$$

The semantics of the intermediate language is given in Fig. 6. When transforming the original program, we keep information on the function definition dependencies as relation $R \subseteq dom(P) \times F$ where F denote the set of functions (including local functions). $R(f, g)$ means that f is a top-level function and function g is defined in the body of f in the original program. We write $F_{\mathtt{top}}$ for the top-level function of the original program P, i.e., $dom(P)$. We assume that the translated program contains a distinguished function symbol $\mathtt{main} \in \{f_1, \ldots, f_n\}$ whose simple type is $\mathbf{int} \to \mathbf{int}$, and `main` does not use its argument. The transformation from the target language to the intermediate language can be defined as a combination of CPS transformation and λ-lifting.

The operational semantics is given by Fig. 6. This semantics is used just for collecting information on which branch is taken, and which function is called in each application. The reduction is labeled with

$$\rho \in \{\varepsilon\} \cup \{(\mathbf{br}, b) \mid b \in \{\mathbf{then}, \mathbf{else}\}\} \cup \{(\mathbf{sp}, \pi) \mid \pi : (L \times \{\mathbf{then}, \mathbf{else}\})^*\}$$

for recording which branch has been taken and which modular counterexample has been used. The label $(\mathbf{br}, \mathbf{then})$ ($(\mathbf{br}, \mathbf{else})$, resp.) represents that the then-branch (else-branch, resp.) is taken. The label $(\mathbf{sp}, \pi)$ represents that the modular counterexample π is used for the top-level function f on the application of f. The evaluation ignores base values, as $\mathbf{assume}\,(v)\,; e$ and $\mathbf{let}\ x = *_{\mathbf{int}}\ \mathbf{in}\ e$ are reduced to e. In the application of top-level function f, the function is duplicated by the function $\mathbf{Spawn}(D, R, f, \pi)$ with respect to modular counterexample π.

$$\frac{f \notin F_{\mathtt{top}} \qquad (f\,\widetilde{x} = e_1 \,\square^l\, e_2) \in D}{(D, B, f\,\widetilde{v}) \xrightarrow{\varepsilon}_{R,\Pi} (D, B, [\widetilde{v}/\widetilde{x}](e_1 \,\square^l\, e_2))}$$

$$\frac{\begin{array}{c} f \in F_{\mathtt{top}} \qquad (f\,\widetilde{x} = t) \in D \qquad |\widetilde{x}| = |\widetilde{v}| \\ (D', \pi', f') = \mathbf{Spawn}(D, R, f, \pi) \qquad (f, \pi) \in \Pi \end{array}}{(D, B, f\,\widetilde{v}) \xrightarrow{(\mathbf{sp},\pi)}_{R,\Pi} (D' \cup D, \{\pi'\} \cup B, f'\widetilde{v})}$$

$$\frac{\begin{array}{c} \pi = (\ell_1, b_1) \ldots (\ell_i, b_i) \ldots (\ell_n, b_n) \qquad \ell_i = \ell \qquad \ell_j \neq \ell \quad \text{for } j \in \{1, \ldots, i-1\} \\ \pi' = (\ell_1, b_1) \ldots (\ell_{i-1}, b_{i-1})(\ell_{i+1}, b_{i+1}) \ldots (\ell_n, b_n) \end{array}}{\big(D, \{\pi\} \uplus B, e_1 \,\square^l\, e_2\big) \xrightarrow{(\mathbf{br},b)}_{R,\Pi} (D, \{\pi'\} \uplus B, e_b)}$$

$$(D, B, \mathbf{assume}\,(v)\,; e) \xrightarrow{\varepsilon}_{R,\Pi} (D, B, e)$$

$$(D, B, \mathbf{let}\ x = *_{\mathbf{int}}\ \mathbf{in}\ a) \xrightarrow{\varepsilon}_{R,\Pi} (D, B, e)$$

$$\begin{array}{l} \mathbf{Spawn}(D, R, f, \pi) \stackrel{\text{def}}{=} (D', \pi', f') \\ \quad \text{where } \{g_1, \ldots, g_n\} = \{g \mid R(f, g)\} \\ \qquad \{l_f\} = \Big\{ l \;\Big|\; (f\,\widetilde{x} = e_1 \,\square^\ell\, e_2) \in D \Big\} \\ \qquad \{l_i\} = \Big\{ l \;\Big|\; (g_i\,\widetilde{x} = e_1 \,\square^\ell\, e_2) \in D \Big\} \quad \text{for each } i \in \{1, \ldots, n\} \\ \qquad f', g'_1, \ldots, g'_n, l'_f, l'_1, \ldots, l'_n \text{ are fresh} \\ \qquad \sigma = [f'/f, g'_0/g_0, \ldots, g'_n/g_n] \\ \qquad D' = \Big\{ f\,\widetilde{x} = \sigma e_1 \,\square^\ell\, \sigma e_2 \;\Big|\; (f\,\widetilde{x} = e_1 \,\square^\ell\, e_2) \in D \Big\} \\ \qquad\qquad \cup \Big\{ g_i\,\widetilde{x} = \sigma e_1 \,\square^{l'_i}\, \sigma e_2 \;\Big|\; i \in \{1, \ldots, n\}, (g_i\,\widetilde{x} = e_1 \,\square\, e_2) \in D \Big\} \\ \qquad \pi = (l_{j_1}, b_1) \ldots (l_{j_k}, b_k) \\ \qquad \pi' = (l'_{j_1}, b_1) \ldots (l'_{j_k}, b_k) \end{array}$$

Fig. 7. Operational semantics of the intermediate language with respect to modular counterexamples

We write $(D, B, t) \overset{\rho_1 \cdots \rho_n}{\Longrightarrow}_{R,\Pi} (D', B', t')$ if

$$(D, B, t)\ (\overset{\varepsilon}{\longrightarrow}_{R,\Pi})^*\ \overset{\rho_1}{\longrightarrow}_{R,\Pi}\ (\overset{\varepsilon}{\longrightarrow}_{R,\Pi})^* \cdots$$
$$(\overset{\varepsilon}{\longrightarrow}_{R,\Pi})^*\ \overset{\rho_n}{\longrightarrow}_{R,\Pi}\ (\overset{\varepsilon}{\longrightarrow}_{R,\Pi})^* (D', B', t').$$

To construct a program corresponding to modular counterexamples, we first extract the set of sequences of labels, which can be viewed as a set of ordinary counterexamples. We define the set $\mathfrak{B}$ of sequence of labels from reduction sequences according to Π by

$$\mathfrak{B} = \left\{ \widetilde{\rho} \;\middle|\; (D, \emptyset, \texttt{main}\,\langle\rangle) \overset{\widetilde{\rho}}{\Longrightarrow}_{R,\Pi} (D', B', t') \text{ for some } D',\ B', \text{ and } t' \right\}.$$

We then construct a program from $\mathfrak{B}$ by using function $\mathbf{Construct}(-,-)$, which is defined in Fig. 8. In the figure, $t_{\rho_1 \ldots \rho_n}$ is a term satisfying

$$(D, \emptyset, \texttt{main}\,\langle\rangle) \overset{\rho_1 \cdots \rho_n}{\Longrightarrow}_{R,\Pi} (D', B', t_{\rho_1 \ldots \rho_n})$$

for some D' and B'. We assign an index to each element of $\mathfrak{B}$, and write $I(\widetilde{\rho})$ for the index of $\widetilde{\rho}$. We assume $I(\widetilde{\rho}) \leq I(\widetilde{\rho'})$ if $\widetilde{\rho}$ is a prefix of $\widetilde{\rho'}$. We write $\widetilde{\rho}_j$ for the element of $\mathfrak{B}$ whose index is j.

Finally, by using the refinement (intersection) type inference [23, 21], we infer a refinement type of the constructed program in the context $C_{\emptyset,\tau}$ where τ is the target type, and return the inferred types as type candidates.

$$
\begin{aligned}
&\mathbf{Construct}(\{f_1\,\widetilde{x}_1 = e_{10} \,\square\, e_{11}, \ldots, f_m\,\widetilde{x}_m = e_{m0} \,\square\, e_{m1}\}, \mathfrak{B}) \stackrel{\text{def}}{=} \\
&\quad \{f_i^{(\widetilde{\rho}(\mathbf{br},k))}\ \widetilde{x}_i = [e_{ik}]_{\widetilde{\rho}(\mathbf{br},k)\rho'} \mid i \in \{1,\ldots,m\},\ j \in \{1,\ldots,|\mathfrak{B}|\},\ f_i \notin F, \\
&\qquad \widetilde{\rho}(\mathbf{br},k)\rho' \in \mathfrak{B},\ t_{\widetilde{\rho}(\mathbf{br},k)} \text{ is an application of } f_i\} \\
&\cup \{f_i^{(\widetilde{\rho})}\ \widetilde{x}_i = \langle\rangle \mid i \in \{1,\ldots,m\},\ j \in \{1,\ldots,|\mathfrak{B}|\},\ f_i \notin F, \\
&\qquad t_{\widetilde{\rho}} \text{ is not an application of } f_i\} \\
&\cup \{f_i^{(\widetilde{\rho})}\ \widetilde{x}_i = f_i^{(\widetilde{\rho}(\mathbf{sp},\Pi_1))}\ \widetilde{x}_i \,\square\, \cdots \,\square\, f_i^{(\widetilde{\rho}(\mathbf{sp},\Pi_k))}\ \widetilde{x}_i \mid i \in \{1,\ldots,m\},\ j \in \{1,\ldots,|\mathfrak{B}|\}, \\
&\qquad f_i \in F, \{c_1,\ldots,c_k\} = \{c \mid (f_i,c) \in \Pi\},\ t_{\widetilde{\rho}} \text{ is an application of } f_i\} \\
&\cup \{f_i^{(\widetilde{\rho})}\ \widetilde{x}_i = \langle\rangle \mid i \in \{1,\ldots,m\},\ j \in \{1,\ldots,|\mathfrak{B}|\}, \\
&\qquad f_i \in F, \{c_1,\ldots,c_k\} = \{c \mid (f_i,c) \in \Pi\},\ t_{\widetilde{\rho}} \text{ is not an application of } f_i\} \\
&\cup \left\{\mathtt{main}\langle\rangle = \mathtt{main}^{(\varepsilon)}\langle\rangle\right\}
\end{aligned}
$$

$$
\begin{aligned}
&[\mathbf{assume}\,(v)\,a]_j = \mathbf{assume}\,(v)\,[a]_j \\
&[\mathbf{let}\ x = \mathtt{op}(\widetilde{v})\ \mathbf{in}\ a]_j = \mathbf{let}\ x = \mathtt{op}(\widetilde{v})\ \mathbf{in}\ [a]_j \\
&[\langle\rangle]_j = \langle\rangle \\
&[\mathbf{fail}]_j = \mathbf{fail} \\
&[x]_j = x \\
&[x\ v_1 \cdots v_k]_j = \sharp_j(x)\ {v_1}^{\diamond_{j+1}} \cdots {v_k}^{\diamond_{j+1}} \qquad (k \geq 1) \\
&[f\ v_1 \cdots v_k]_j = f^{(\widetilde{\rho_j})}\ {v_1}^{\diamond_{j+1}} \cdots {v_k}^{\diamond_{j+1}}
\end{aligned}
$$

$$
\begin{aligned}
&c^{\diamond_j} = c \\
&x^{\diamond_j} = x \quad \text{(if } x \text{ is a variable of a base type)} \\
&(x\ \widetilde{v})^{\diamond_j} = \langle\underbrace{\lambda\widetilde{y}.\langle\rangle, \ldots, \lambda\widetilde{y}.\langle\rangle}_{j-1}, \sharp_j(x)(\widetilde{v}^{\diamond_j}), \ldots, \sharp_{|\mathfrak{B}|}(x)(\widetilde{v}^{\diamond_j})\rangle \\
&\qquad \text{(if } x \text{ is a function variable)} \\
&(f\ \widetilde{v})^{\diamond_j} = \langle\underbrace{\lambda\widetilde{y}.\langle\rangle, \ldots, \lambda\widetilde{y}.\langle\rangle}_{j-1}, f^{(\rho_j)}(\widetilde{v}^{\diamond_j}), \ldots, f^{(\rho_{|\mathfrak{B}|})}(\widetilde{v}^{\diamond_j})\rangle
\end{aligned}
$$

Fig. 8. The definition of $\mathbf{Construct}(D, \mathfrak{B})$

B Proof of Lemma 1

Definition 1 (Simulation). *A* simulation *is a family of relations* $\{\mathcal{R}^\kappa\}_\kappa$ *such that* $\mathcal{R}^\kappa$ *is a relation between terms of simple type* κ*, and if* $t_1\ \mathcal{R}^\kappa\ t_2$*, then either* $t_2 \longrightarrow_P^* \mathbf{fail}$ *or the following hold:*

- *If* $t_1 \longrightarrow_P^* n$*, then* $t_2 \longrightarrow_P^* n$*.*
- *If* κ *is of the form* $\kappa_1 \to \kappa_2$ *and* $t_1 \longrightarrow_P^* \mathbf{fix}(f, \lambda x.\, t_1')$*, then there exists* t_2' *such that*
  - $t_2 \longrightarrow_P^* \mathbf{fix}(f, \lambda x.\, t_2')$ *and*
  - $[v_1/x][\mathbf{fix}(f, \lambda x.\, t_1')/f]t_1'\ \mathcal{R}^{\kappa_2}\ [v_2/x][\mathbf{fix}(f, \lambda x.\, t_2')/f]t_2'$ *for any values* v_1 *and* v_2 *such that* $v_1\ \mathcal{R}^{\kappa_1}\ v_2$*.*
- *If* $t_1 \longrightarrow_P^* \mathbf{fail}$*, then* $t_2 \longrightarrow_P^* \mathbf{fail}$*.*

We define $\{\lesssim^\kappa\}_\kappa$ *as the greatest simulation. For open terms* t_1 *and* t_2*, we also write* $t_1 \lesssim^\kappa t_2$ *if, for some simple type environment* $\Gamma = x_1 : \kappa_1, \ldots, x_n : \kappa_n$*,*

- t_1 *and* t_2 *have simple type* κ *under* Γ*, and*
- $[v_1/x_1, \ldots, v_n/x_n]t_1 \lesssim^\kappa [v_1/x_1, \ldots, v_n/x_n]t_2$ *for any* $v_1, \ldots, v_n$ *such that* v_i *has type* κ_i *for each* i*.*

Definition 2 (Size of type). *The size* $size(\tau)$ *(*$size_\wedge(\sigma)$*, resp.) of type* τ *(*σ*, resp.) is defined as follows:*

$$
\begin{aligned}
&size(\{x : \mathbf{int} \mid \phi\}) = 1\\
&size((x : \sigma) \to \tau) = 1 + size_\wedge(\sigma) + size(\tau)\\
&size_\wedge(\{\tau_1, \ldots, \tau_n\}_\kappa) = 1 + size(\tau_1) + \cdots + size(\tau_n).
\end{aligned}
$$

Lemma 3. *Suppose* $t_1 \lesssim^\kappa t_2$*.*

- *If* $\models^P t_2 : \tau$*, then* $\models^P t_1 : \tau$*.*
- *If* $\models_\wedge^P t_2 : \sigma$*, then* $\models_\wedge^P t_1 : \sigma$*.*

Proof. By induction on $size(\tau)$ and $size(\sigma)$. Suppose $t_1 \lesssim^\kappa t_2$ and $\models^P t_2 : \tau$. If $t_1 \longrightarrow_P^* \mathbf{fail}$, by the assumption $t_1 \lesssim^\kappa t_2$, we have $t_2 \longrightarrow_P^* \mathbf{fail}$, which contradicts $\models^P t_2 : \tau$. We show $\models_{\mathrm{v}}^P v : \tau$ for any v such that $t_1 \longrightarrow_P^* v$.

Case $v = n$ and $\models_{\mathrm{v}}^P v : \tau$: By the assumption $t_1 \lesssim^\kappa t_2$, we have $t_2 \longrightarrow_P^* n$ and $\models^P n : \tau$, as required.

Case $v = \mathbf{fix}(f, \lambda x.\, t_1')$ and $\models_{\mathrm{v}}^P v : (x : \sigma_1) \to \tau_2$: We have $\kappa = \kappa_1 \to \kappa_2$ for some κ_1 and κ_2. By the assumption $t_1 \lesssim^\kappa t_2$, there exists t_2' such that $t_2 \longrightarrow_P^* \mathbf{fix}(f, \lambda x.\, t_2')$ and $[v_1/x][\mathbf{fix}(f, \lambda x.\, t_1')/f]t_1' \lesssim^{\kappa_2} [v_2/x][\mathbf{fix}(f, \lambda x.\, t_2')/f]t_2'$ for any values v_1 and v_2 such that $v_1 \lesssim^{\kappa_1} v_2$. By the assumption $\models^P t_2 : \tau$, we have $\models^P [v_2/x][\mathbf{fix}(f, \lambda x.\, t_2')/f]t_2' : [v_2/x]\tau_2$ for any v_2 such that $\models_{\mathrm{v},\wedge}^P v_2 : \sigma_1$. Let v' be a value such that $\models_{\mathrm{v},\wedge}^P v' : \sigma_1$. Since $v' \lesssim^{\kappa_1} v'$, we get

$$
\begin{aligned}
&\models^P [v'/x][\mathbf{fix}(f, \lambda x.\, t_2')/f]t_2' : [v'/x]\tau_2\\
\Rightarrow\ &\models^P [v'/x][\mathbf{fix}(f, \lambda x.\, t_1')/f]t_1' : [v'/x]\tau_2 \qquad \text{(by I.H.)}\\
\Rightarrow\ &\models^P v\, v' : [v'/x]\tau_2 \qquad (\text{since } v\, v' \preceq_P [v'/x][\mathbf{fix}(f, \lambda x.\, t_1')/f]t_1').
\end{aligned}
$$

Thus, we obtain $\models_{\mathrm{v}}^P v : \tau$.

Case $\models_{\mathrm{v},\wedge}^P v : \sigma$: By I.H.

Lemma 4. *If $v_1 \lesssim^{\kappa} v_2$, then $[v_2/x]\tau = [v_1/x]\tau$.*

Proof. If $\kappa = \mathbf{int}$, then we have $v_1 = v_2$. Therefore, we get $[v_2/x]\tau = [v_1/x]\tau$. If κ is a function type, since a variable of a function type cannot occur in τ, we have $[v_2/x]\tau = \tau = [v_1/x]\tau$.

Lemma 5. *If $t_1 \lesssim^{\kappa_1 \to \kappa_2} t_2$ and $t_1' \lesssim^{\kappa_1} t_2'$, then $t_1\, t_1' \lesssim^{\kappa_2} t_2\, t_2'$.*

Proof. Suppose $t_1\, t_1' \longrightarrow_P^* v$. We have

- $t_1 \longrightarrow_P^* \mathbf{fix}(f, \lambda x.\, t_3)$,
- $t_1' \longrightarrow_P^* v_1$, and
- $[v_1/x][\mathbf{fix}(f, \lambda x.\, t_3)/f]t_3 \longrightarrow_P^* v$

for some t_3 and v_1. By the assumption that $t_1' \lesssim^{\kappa_1} t_2'$, we have $v_1 \lesssim^{\kappa_1} v_2$ for some v_2 such that $t_2' \longrightarrow_P^* v_2$. Therefore, by the assumption that $t_1 \lesssim^{\kappa_1 \to \kappa_2} t_2$, we get $[v_1/x][\mathbf{fix}(f, \lambda x.\, t_3)/f]t_3 \lesssim^{\kappa_2} [v_2/x][\mathbf{fix}(f, \lambda x.\, t_4)/f]t_4$ for some t_4 such that $t_2 \longrightarrow_P^* \mathbf{fix}(f, \lambda x.\, t_4)$.

Lemma 6. *If $v \lesssim^{\kappa} \alpha_\wedge(\sigma)$, then there exists v' such that $\alpha_\wedge(\sigma) \longrightarrow_P^* v'$ and $v \lesssim^{\kappa} v'$.*

Proof. By case analysis on σ.

Lemma 7. *Suppose $FV(\tau) = \{x\}$ and $[v/x]\tau$ is a valid type, i.e., predicates in $[v/x]\tau$ are well-typed and have type* **int**. *Then, $[v/x]\alpha(\tau) = \alpha([v/x]\tau)$, and $[v/x]\beta(v' : \tau) = \beta(v' : [v/x]\tau)$.*

Proof. By induction on the size of $ST(\tau)$.

Lemma 8. *For any type τ and σ, the following holds.*

1. $\models^P \alpha(\tau) : \tau$.
2. $\models^P_\wedge \alpha_\wedge(\sigma) : \sigma$.
3. *If* $\models^P_{\mathrm{v}} v : \tau$, *then* $\beta(v : \tau) \preceq_P \mathbf{true}$.
4. *If* $\models^P_{\mathrm{v},\wedge} v : \sigma$, *then* $\beta_\wedge(v : \sigma) \preceq_P \mathbf{true}$.

Proof. By induction on $size(\tau)$ and $size_\wedge(\sigma)$.

Case $\tau = \{x : \mathbf{int} \mid \phi\}$: By the definition of $\alpha(-)$, we have

$$\alpha(\tau) = \mathbf{let}\ x = *_{\mathbf{int}}\ \mathbf{in}\ \mathbf{assume}(\phi);\ x.$$

We show that $\models^P \mathbf{assume}([n/x]\phi);\, n : \tau$ for any integer n. Since ϕ does not include applications and $*_{\mathbf{int}}$, there exists a unique v such that $[n/x]\phi \preceq_P v$. If $v = \mathbf{true}$, since $\models^P_{\mathrm{p}} [n/x]\phi$ holds, we obtain $\models^P_{\mathrm{v}} n : \tau$. If $v \neq \mathbf{true}$, since $\mathbf{assume}([n/x]\phi) \not\longrightarrow_P^* v'$ for any v', we have $\models^P \mathbf{assume}([n/x]\phi);\, n : \tau$ as required. Suppose $\models^P_{\mathrm{v}} n' : \tau$ for some integer n'. $\beta(n' : \tau) = [n'/x]\phi \preceq_P \mathbf{true}$ follows from the definition of $\models^P_{\mathrm{v}} n' : \tau$.

Case $\tau = (x : \sigma_1) \to \tau_2$: By the definition of $\alpha(-)$, we have

$$\alpha(\tau) = \lambda x.\, \mathbf{if}\ * \vee \beta(x : \sigma_1)\ \mathbf{then}\ \alpha(\tau_2)$$

We show that $\models^P \alpha(\tau)\, v : [v/x]\tau_2$ for any v such that $\models^P_{\mathrm{v},\wedge} v : \sigma_1$. We get $\beta(v : \sigma_1) \preceq_P \mathbf{true}$ by I.H. Therefore, we have $\alpha(\tau)\, v \preceq_P \alpha([v/x]\tau_2)$ by Lemma 7. Since $\models^P \alpha([v/x]\tau_2) : [v/x]\tau_2$ by I.H., we get $\models^P \alpha(\tau)\, v : [v/x]\tau_2$ as required. We next show that $\beta(v : \tau) \preceq_P \mathbf{true}$ for any v such that $\models^P_{\mathrm{v}} v : \tau$. By the definition of $\beta(-)$, we have

$$\beta(v : \tau) = \mathbf{let}\ x = \alpha(\sigma_1)\ \mathbf{in}\ \mathbf{let}\ r = v\, x\ \mathbf{in}\ \beta(r : \tau_2).$$

Suppose $\alpha(\sigma_1) \longrightarrow^*_P v'$, $v\, v' \longrightarrow^*_P v''$, and

$$\beta(v : \tau) \longrightarrow^*_P \mathbf{let}\ r = v\, v'\ \mathbf{in}\ [v'/x]\beta(r : \tau_2) \longrightarrow^*_P [v'/x]\beta(v'' : \tau_2).$$

Since $\models^P_{\wedge} \alpha(\sigma_1) : \sigma_1$ by I.H., we have $\models^P_{\mathrm{v},\wedge} v' : \sigma_1$ and $\models^P_{\mathrm{v}} v'' : [v'/x]\tau_2$. By I.H., we get $\beta(v'' : [v'/x]\tau_2) \preceq_P \mathbf{true}$, and hence, $[v'/x]\beta(v'' : \tau_2) \preceq_P \mathbf{true}$ by Lemma 7.

Case $\sigma = \emptyset_\kappa$: Trivial.

Case $\sigma = \{\tau_1, \ldots, \tau_n\}_{\mathbf{int}}$: By I.H.

Case $\sigma = \{\tau_1, \ldots, \tau_n\}_{\kappa_1 \to \kappa_2}$: Follows from the following fact: For any v, $\tau :: \kappa$, and $\sigma' = \{\tau'_1, \ldots, \tau'_n\}_\kappa$, if $\models^P_{\mathrm{v},\wedge} v : \sigma'$, then $\models^P_{\wedge} \mathit{wrap}(\tau, v) : \{\tau, \tau'_1, \ldots, \tau'_n\}_\kappa$. We show the fact by induction on $\mathit{size}(\sigma)$ under the restriction $\mathit{size}(\sigma') < \mathit{size}(\sigma)$.

Case $\tau = \{x : \mathbf{int} \mid \phi\}$: Suppose $\mathit{wrap}(\tau, v) = \mathbf{assume}([v/x]\phi);\, v \longrightarrow^*_P v'$. Since we have $v' = v$ and $\models^P_{\mathrm{v}} v : \{x : \mathbf{int} \mid \phi\}$, we obtain $\models^P_{\mathrm{v}} v' : \{\{x : \mathbf{int} \mid \phi\}, \tau'_1, \ldots, \tau'_n\}$.

Case $\tau = (x : \sigma_0) \to \tau_0$: We first show $\models^P \mathit{wrap}(\tau, v) : \tau$, i.e., $\models^P \mathit{wrap}(\tau, v)\, v' : [v'/x]\tau_0$ for any v' such that $\models^P_{\mathrm{v},\wedge} v' : \sigma_0$. Let t_0 be $\mathit{wrap}(\tau, v)\, v'$. Suppose $t_0 \longrightarrow^*_P v''$. By I.H. of the lemma, we have $\beta_\wedge(v' : \sigma_0) \preceq_P \mathbf{true}$. If $f\, x$ fails, i.e., the reduction is of the form

$$t_0 \longrightarrow^*_P E[\mathbf{let}\ r = v\, v'\ \mathbf{in}\ \mathit{wrap}(\tau_0, r)] \longrightarrow^*_P E[\mathbf{fail}],$$

then $\alpha([v'/x]\tau_0) \preceq_P v''$. Therefore, we get $\models^P_{\mathrm{v}} v'' : [v'/x]\tau_0$ by I.H. of the lemma. If $f\, x$ does not fail, i.e., $t_0 \longrightarrow^*_P \mathit{wrap}(\tau_0, v_r)$ for some v_r, then we obtain $\models^P_{\mathrm{v}} v'' : [v'/x]\tau_0$ by I.H. Suppose $\sigma' = \{(x_1 : \sigma_1) \to \tau'_1, \ldots, (x_n : \sigma_n) \to \tau'_n\}$ We next show $\models^P_{\wedge} \mathit{wrap}(\tau, v) : \sigma'$, i.e., $\models^P \mathit{wrap}(\tau, v) : (x_i : \sigma_i) \to \tau'_i$ for each $i \in \{1, \ldots, n\}$. Suppose $\models^P_{\mathrm{v},\wedge} v' : \sigma_i$ and $\mathit{wrap}(\tau, v)\, v' \longrightarrow^*_P v''$. We show $\models^P_{\mathrm{v}} v'' : [v'/x_i]\tau'_i$. If the then-branch is taken, since $v\, v'$ (i.e., $f\, x$) does not fail, then the reduction is of the form $\mathit{wrap}(\tau, v)\, v' \longrightarrow^*_P \mathit{wrap}(v'', [v'/x_i]\tau'_i)$. Therefore, we obtain $\models^P_{\mathrm{v}} v'' : [v'/x_i]\tau'_i$ by I.H. The case of the else-branch is straightforward.

Lemma 9. *Let i be an integer and v be a value of simple type $ST(\tau) = ST(\sigma)$.*

- *Suppose $t \lesssim^{ST(\tau')} \alpha(\tau')$ for any t and τ' such that $\models^P t : \tau'$ and $\mathit{size}(\tau') < i$. If $\not\models^P_{\mathrm{v}} v : \tau$ and $\mathit{size}(\tau) = i$, then $\beta(v : \tau) \longrightarrow^*_P \mathbf{false}$ or $\beta(v : \tau) \longrightarrow^*_P \mathbf{fail}$.*
- *Suppose $t \lesssim^{ST(\sigma')} \alpha_\wedge(\sigma')$ for any t and σ' such that $\models^P_{\wedge} t : \sigma'$ and $\mathit{size}(\sigma') < i$. If $\not\models^P_{\mathrm{v},\wedge} v : \sigma$ and $\mathit{size}(\sigma) = i$, then $\beta_\wedge(v : \sigma) \longrightarrow^*_P \mathbf{false}$ or $\beta_\wedge(v : \sigma) \longrightarrow^*_P \mathbf{fail}$.*

Proof. By induction on i.

Case $\tau = \{x : \mathbf{int} \mid \phi\}$: We have $v = n$ for some n. By the assumption that $\not\models^P_{\mathrm{v}} v : \tau$, $[v/x]\phi \longrightarrow^*_P \mathbf{false}$.

Case $\tau = (x : \sigma_1) \to \tau_2$: We have $v = \mathbf{fix}(f, \lambda x.\, t)$ for some t. Since $\not\models^P_{\mathrm{v}} v : (x : \sigma_1) \to \tau_2$, there exists v' such that $\models^P_{\mathrm{v},\wedge} v' : \sigma_1$ and $\not\models^P v\, v' : [v'/x]\tau_2$. By the assumption and $\mathit{size}(\sigma_1) < \mathit{size}(\tau) = i$, we have $v' \lesssim^{ST(\sigma_1)} \alpha_\wedge(\sigma_1)$. Hence, we get $v\, v' \lesssim^{ST(\tau_2)} v\, \alpha_\wedge(\sigma_1)$ by Lemma 5. Therefore, by Lemma 3, we get $\not\models^P v\, \alpha_\wedge(\sigma_1) : [v'/x]\tau_2$, i.e., there exists v_1 and a such that $\alpha_\wedge(\sigma_1) \longrightarrow^*_P v_1$, $v\, v_1 \longrightarrow^*_P a$, and $\not\models^P a : [v'/x]\tau_2$. If $a = \mathbf{fail}$, then we obtain $\beta(v : \tau) \longrightarrow^*_P \mathbf{fail}$. If $a = v_2$ for some v_2, since

$$\beta(v : \tau) \longrightarrow^*_P \mathbf{let}\ r = v\, v_1\ \mathbf{in}\ \beta(r : [v'/x]\tau_2) \longrightarrow^*_P \beta(v_2 : [v'/x]\tau_2)$$

we get $\beta(v : \tau) \longrightarrow^*_P \mathbf{false}$ or $\beta(v : \tau) \longrightarrow^*_P \mathbf{fail}$ by I.H.

Case $\sigma = \{\tau_1, \ldots, \tau_n\}$: By I.H.

Lemma 10. *The following holds.*

- *If* $\models^P t : \tau$, *then* $t \lesssim^{ST(\tau)} \alpha(\tau)$.
- *If* $\models^P_\wedge t : \sigma$, *then* $t \lesssim^{ST(\sigma)} \alpha_\wedge(\sigma)$.

Proof. By induction on $\mathit{size}(\tau)$ and $\mathit{size}(\sigma)$.

Case $\tau = \{x : \mathbf{int} \mid \phi\}$ and $t \longrightarrow^*_P \mathbf{fail}$: Contradicts the assumption.

Case $\tau = \{x : \mathbf{int} \mid \phi\}$ and $t \longrightarrow^*_P n$: Since $\models^P_{\mathrm{p}} [n/x]\phi$ and

$$\alpha(\tau) = \mathbf{let}\ x = *_{\mathbf{int}}\ \mathbf{in}\ \mathbf{assume}(\phi);\, x,$$

we get $\alpha(\tau) \longrightarrow^*_P n$. Therefore, we obtain $t \lesssim^{\mathbf{int}} \alpha(\tau)$.

Case $\tau = (x : \sigma_1) \to \tau_2$ and $t \longrightarrow^*_P \mathbf{fail}$: Contradicts the assumption.

Case $\tau = (x : \sigma_1) \to \tau_2$ and $t \longrightarrow^*_P \mathbf{fix}(f, \lambda x.\, t')$: Let $\kappa_1 = ST(\sigma_1)$ and $\kappa_2 = ST(\tau_2)$. We show that there exists t_1 such that

- $\alpha(\tau) \longrightarrow^*_P \mathbf{fix}(f, \lambda x.\, t_1)$ and
- $[v_1/x][\mathbf{fix}(f, \lambda x.\, t')/f]t' \lesssim^{\kappa_2} [v_2/x][\mathbf{fix}(f, \lambda x.\, t_1)/f]t_1$ for any values v_1 and v_2 such that $v_1 \lesssim^{\kappa_1} v_2$.

Let t_1 be $\mathbf{if} * \vee \beta(x : \sigma_1)\ \mathbf{then}\ \alpha(\tau_2)\ \mathbf{else}\ \alpha(\kappa_2)$, then $\alpha(\tau) = \mathbf{fix}(f, \lambda x.\, t_1)$. If $\models^P_{\mathrm{v},\wedge} v_1 : \sigma_1$, then we have $\models^P [v_1/x][\mathbf{fix}(f, \lambda x.\, t')/f]t' : [v_1/x]\tau_2$. Since

$$[v_2/x][\mathbf{fix}(f, \lambda x.\, t_1)/f]t_1 \longrightarrow^*_P [v_2/x]\alpha(\tau_2) = \alpha([v_2/x]\tau_2) = \alpha([v_1/x]\tau_2)$$

by Lemmas 7 and 4, we get

$$[v_1/x][\mathbf{fix}(f, \lambda x.\, t')/f]t' \lesssim^{\kappa_2} [v_2/x][\mathbf{fix}(f, \lambda x.\, t_1)/f]t_1$$

by I.H. If $\not\models^P_{\mathrm{v},\wedge} v_1 : \sigma_1$, then we have $\not\models^P_{\mathrm{v},\wedge} v_2 : \sigma_1$ by Lemma 3, and hence, $\beta(v_2 : \sigma_1) \longrightarrow^*_P \mathbf{false}$ or $\beta(v_2 : \sigma_1) \longrightarrow^*_P \mathbf{fail}$ by Lemma 9. Since $[v_2/x][\mathbf{fix}(f, \lambda x.\, t_1)/f]t_1 \longrightarrow^*_P \mathbf{fail}$, we obtain

$$[v_1/x][\mathbf{fix}(f, \lambda x.\, t')/f]t' \lesssim^{\kappa_2} [v_2/x][\mathbf{fix}(f, \lambda x.\, t_1)/f]t_1$$

as required.

Lemma 11. *$\models^P_{\mathrm{v}} v_1 : (x:\sigma) \to \tau$ if and only if $\models^P v_1\, v_2 : [v_2/x]\tau$ for any v_2 such that $\alpha_\wedge(\sigma) \longrightarrow^*_P v_2$.*

Proof. "Only-if" direction: Obvious from (2) of Lemma 8.

"If" direction: Suppose $\models^P v_1\, v_2 : [v_2/x]\tau_2$ for any v_2 such that $\alpha(\sigma) \longrightarrow^*_P v_2$. We show that $\models^P v_1\, v_2' : [v_2'/x]\tau_2$ for any v_2' such that $\models^P_{\mathrm{v},\wedge} v_2' : \sigma$. We have $v_2' \lesssim^{ST(\sigma)} \alpha(\sigma)$ by Lemma 10, and hence, by Lemma 6, there exists v_2'' such that $\alpha(\sigma) \longrightarrow^*_P v_2''$ and $v_2' \lesssim v_2''$. By the assumption, we get $\models^P v_1\, v_2'' : [v_2''/x]\tau_2$. Therefore, we obtain $\models^P v_1\, v_2' : [v_2'/x]\tau_2$ by Lemmas 3 and 4.

Proof (Proof of Lemma 1). Let

- $\Gamma = x_1 : \sigma_1, \ldots, x_m : \sigma_m$ and
- $\tau = (x_{m+1} : \sigma_{m+1}) \to \cdots \to (x_n : \sigma_n) \to \{r : \mathbf{int} \mid \phi\}$.

By the definition of $(\models^P)$ and the semantics,

$$
\begin{aligned}
& \Gamma \models^P t : \tau \\
\iff\ & \models^P \lambda x_1. \ldots \lambda x_m.\, t : (x_1 : \sigma_1) \to \cdots \to (x_n : \sigma_n) \to \{r : \mathbf{int} \mid \phi\} \\
\iff\ & \forall v_1, \ldots, v_n. \bigwedge_{i \in \{1,\ldots,n\}} \alpha_\wedge\left([v_j/x_j]_{j\in\{1,\ldots,i-1\}}\sigma_i\right) \longrightarrow^*_P v_i \Rightarrow \\
& \models^P t\, v_{m+1} \ldots v_n : \left\{r : \mathbf{int} \;\middle|\; [v_j/x_j]_{j\in\{1,\ldots,n\}}\phi\right\} \\
& \text{(by Lemma 11)} \\
\iff\ & \forall v_1, \ldots, v_n. \bigwedge_{i \in \{1,\ldots,n\}} \alpha_\wedge\left([v_j/x_j]_{j\in\{1,\ldots,i-1\}}\sigma_i\right) \longrightarrow^*_P v_i \Rightarrow \\
& \forall a.\, t\, v_{m+1} \ldots v_n \longrightarrow^*_P a \ \Rightarrow \models^P a : \left\{r : \mathbf{int} \;\middle|\; [v_j/x_j]_{j\in\{1,\ldots,n\}}\phi\right\} \\
\iff\ & \forall v_1, \ldots, v_n. \bigwedge_{i \in \{1,\ldots,n\}} \alpha_\wedge\left([v_j/x_j]_{j\in\{1,\ldots,i-1\}}\sigma_i\right) \longrightarrow^*_P v_i \Rightarrow \\
& \forall a.\, t\, v_{m+1} \ldots v_n \longrightarrow^*_P a \Rightarrow \\
& a \neq \mathbf{fail} \wedge \models^P \mathbf{assert}([a/r][v_j/x_j]_{j\in\{1,\ldots,n\}}\phi) : \mathbf{int} \\
\iff\ & \forall v_1, \ldots, v_n. \bigwedge_{i \in \{1,\ldots,n\}} \alpha_\wedge\left([v_j/x_j]_{j\in\{1,\ldots,i-1\}}\sigma_i\right) \longrightarrow^*_P v_i \Rightarrow \\
& \models^P \mathbf{let}\ r = t\, v_{m+1} \ldots v_n\ \mathbf{in}\ \mathbf{assert}([v_j/x_j]_{j\in\{1,\ldots,n\}}\phi) : \mathbf{int} \\
\iff\ & \models^P \mathbf{let}\ x_1 = \alpha_\wedge(\sigma_1)\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_n = \alpha_\wedge(\sigma_n)\ \mathbf{in} \\
& \quad \mathbf{let}\ r = t\, x_1 \ldots x_n\ \mathbf{in}\ \mathbf{assert}(\phi) : \mathbf{int} \\
\iff\ & \models^P \mathbf{let}\ x_1 = \alpha_\wedge(\sigma_1)\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_m = \alpha_\wedge(\sigma_m)\ \mathbf{in\ let}\ r_m = t\ \mathbf{in} \\
& \quad \mathbf{assert}(\mathbf{let}\ x_{m+1} = \alpha_\wedge(\sigma_{m+1})\ \mathbf{in\ let}\ r_{m+1} = r_m\, x_{m+1}\ \mathbf{in}\ \ldots \\
& \qquad \mathbf{let}\ x_n = \alpha_\wedge(\sigma_n)\ \mathbf{in\ let}\ r_{m+1} = r_m\, x_{m+1}\ \mathbf{in}\ \phi) : \mathbf{int} \\
& \text{(since } \alpha_\wedge(\sigma_i) \text{ does not fail)} \\
=\ & \models^P \mathbf{let}\ x_1 = \alpha_\wedge(\sigma_1)\ \mathbf{in}\ \ldots\ \mathbf{let}\ x_m = \alpha_\wedge(\sigma_m)\ \mathbf{in\ let}\ r_m = t\ \mathbf{in}
\end{aligned}
$$

$$
\begin{aligned}
& \beta\,(r_m : \tau) : \mathbf{int} \\
= \;& \models^P C_{\Gamma,\tau}[t] : \mathbf{int} \\
\iff \;& C_{\Gamma,\tau}[t] \not\longrightarrow_P^* \mathbf{fail}
\end{aligned}
$$