Automatic Total Correctness Verification of Higher-Order Functional Programs with Algebraic Data Types

Kodai Hashimoto Yuki Satake Sho Torii Hiroshi Unno
University of Tsukuba
{kodai,satake,sho,uhiro}@logic.cs.tsukuba.ac.jp

Abstract

We present a refinement type system for total correctness verification (i.e., partial correctness and termination verification) of ML-like higher-order functional programs with algebraic data types. To fully automate verification, we propose a novel refinement type inference method based on Horn constraint solving that can infer inductive invariants and ranking functions used respectively for partial correctness and termination verification. To our knowledge, our method is the first to automatically infer invariants and ranking functions over not only numerical values but also over higher-order functions and algebraic data structures. To this end, we propose novel techniques to lift existing invariant and ranking function synthesis methods for numerical values to those for functions and data structures. We have implemented a total correctness verification tool for the OCaml functional language based on the proposed method, and confirmed that it can verify, with a low annotation burden, the total correctness of widely used standard libraries of the OCaml language as well as tricky higher-order recursive functions that were used as benchmarks in previous studies.

1. Introduction

Recent years have witnessed intense development of fully- or semi-automated methods for path-sensitive verification of higher-order functional programs [20, 22, 31, 38, 40, 42, 43]. Interestingly, most of the methods are based on refinement type systems [8, 44, 45], where specifications and invariants of the functions of a given program are expressed as *refinement types*, namely dependent types equipped with first-order formulas of some efficiently decidable underlying logic such as the quantifier-free theory of linear integer arithmetic (QFLIA). Thus, as summarized in Table 1, refinement type systems are considered as axiomatic semantics for higher-order functional programs (cf. Hoare logics for imperative programs), and an automated verification of higher-order functional programs is often formalized as refinement type inference via Horn constraint generation [8, 40] (cf. weakest precondition generation for imperative programs).

Refinement types are particularly amenable to automatic checking and inference because typing constraints can be expressed as logical formulas (in the form of Horn clauses with predicate variables) as with verification conditions in Hoare logics, and therefore well-integratable with off-the-shelf SMT solvers [7] for type checking and Horn constraint based invariant synthesizers such as [14, 15, 17, 26, 32, 38–41] for type inference (see [12] for an overview of the Horn constraint based approach to verification, which is actually not limited to functional programs). In fact, as shown in Table 1, there have been proposed fully- or semi-automated refinement type inference methods [20, 22, 31, 38, 40, 42].

The refinement type systems underlying these automated methods, however, are only for partial correctness verification and cannot be used to verify the termination of higher-order functional programs. Though there are refinement types systems for total correctness verification (i.e., partial correctness and termination verification) of functional programs [43, 44], they are not fully automated and sometimes put on users a heavy burden of annotating their programs with complex invariants and ranking functions (i.e., termination arguments). In Table 1, "fully-automated" indicates that a refinement type inference method is integrated with ranking function synthesizers such as [23, 28, 30] for termination verification and invariant synthesizers for partial correctness verification. "Semi-automated" indicates that an inference method relies on hints of invariants and ranking functions that are user-supplied and/or syntactically extracted from the program.

There exist other approaches to fully-automated termination verification of higher-order functional programs such as transition invariants [23], size-change analysis [19, 34–36], and term rewriting [11]. These methods, however, still have a room for improvements with respect to analysis precision and efficiency, as we will discuss in Section 8. This situation is in stark contrast to termination verification of first-order programs, where much work has been done on automated termination verification [4, 5, 10, 16, 24, 29].

To overcome the situation, this paper presents a novel refinement type system for *total correctness* verification of ML-like (i.e., higher-order, typed, and strict) functional programs with *algebraic data types* (ADTs). The main advantage of the system is that it is carefully designed to-

	Higher-Order Functional			Imperative
Axiomatic Semantics (Partial Correctness)	Refinement type systems (partial correctness)			Hoare logic (partial correctness)
	Fully- [22, 38, 40, 42]	**Semi-** [20, 31]	**Non-automated** [8, 45]	
Axiomatic Semantics (Total Correctness)	Refinement type systems (total correctness)			Hoare logic (total correctness)
	Fully- *this paper*	**Semi-** [43]	**Non-automated** [44]	
Predicate Transformer Semantics	Horn constraint generation [21, 40]			Weakest precondition generation

Table 1. Semantics for verification of imperative and higher-order functional programs

ward fully-automated type inference. In fact, we present a Horn constraint based type inference method for the system, which automatically infers necessary invariants and ranking functions over higher-order recursive functions and algebraic data structures as well as integers. The inference method internally generates and solves typing constraints expressed as Horn clause and well-foundedness constraints.[1] We have implemented a type checking and inference tool based on the proposed method, and confirmed that it can verify, with a modest annotation burden, the total correctness of widely used OCaml standard libraries including List and Map modules and small but tricky higher-order recursive functions obtained from previous studies that require precise analysis of the higher-order control flow and/or synthesis of complex ranking functions.

The rest of the paper is organized as follows. Section 2 gives an informal overview of our type system and type inference method. Section 3 explains our target ML-like language $\mathcal{L}$ with ADTs. Section 4 formalizes a refinement type system $\vdash_P$ for partial correctness verification of programs in the language $\mathcal{L}$. We then extend $\vdash_P$ to obtain our refinement type system $\vdash_T$ for total correctness verification in Section 5. In Section 6, we present our type inference method for the system $\vdash_T$. We report on our implementation and experiment results in Section 7. We compare our method with related work in Section 8 and conclude the paper in Section 9.

2. Overview

This section gives an informal overview of our refinement type system $\vdash_T$ and a type inference method of $\vdash_T$ for total correctness verification of ML-like higher-order functional programs with ADTs. Example programs are written in OCaml syntax.

Recursive functions on integers As a first example, let us consider a simple recursive function on integers

```
let rec sum x = if x<=0 then 0 else x+sum(x-1)
```

and its total correctness specification "for any integer argument n, the evaluation of `sum` n always terminates and the return value is not less than the argument n". As in ordinary refinement type systems, our system expresses the specification as a refinement type

$$\tau_{\texttt{sum}} \triangleq (x : \texttt{int}) \to \{y : \texttt{int} \mid y \geq x\}\,.$$

Here, y represents the return value of `sum` when applied to the argument x. The total correctness verification comprises (1) the partial correctness and (2) termination verification.

Our method automatically verifies (1) the partial correctness of `sum` by inferring its inductive invariant sufficient to establish the specification $\tau_{\texttt{sum}}$ and (2) the termination of the evaluation of `sum` n for any argument n of the type `int`, by inferring a ranking function $r_{\texttt{sum}}(x) \triangleq x$ that witnesses the well-foundedness of `sum`'s *recursion relation*

$$Rec_{\texttt{sum}} \triangleq \{(x, x-1) \mid x > 0\}\,,$$

which represents the relational invariant between the argument x of a call to `sum` and the argument $x-1$ of its immediate recursive call in the else branch, where $x > 0$ always holds. A ranking function r represents the well-founded relation $wfrel(r) \triangleq \{(x, x') \mid r(x) > r(x') \geq 0\}$. Note that $r_{\texttt{sum}}$ witnesses the well-foundedness of $Rec_{\texttt{sum}}$ because $Rec_{\texttt{sum}} \subseteq wfrel(r_{\texttt{sum}})$, and consequently the termination of `sum` is concluded because an infinite recursive call sequence of `sum` would cause an infinite descent in the argument values with respect to the well-founded relation $Rec_{\texttt{sum}}$, which is a contradiction.

Our refinement type inference method generates Horn clause and well-foundedness constraints on predicate variables that represent inductive invariants and recursion relations of the functions of the given program. For the function `sum`, we obtain the following set $\mathcal{W}_{\texttt{sum}}$ of Horn clause and well-foundedness constraints:

$$\left\{ \begin{array}{l} P(x,0) \Leftarrow x \leq 0, \quad P(x, x+y) \Leftarrow P(x-1, y) \wedge x > 0, \\ Rec_{\texttt{sum}}(x, x') \Leftarrow x' = x - 1 \wedge x > 0, \\ y \geq x \Leftarrow P(x, y), \quad WF(Rec_{\texttt{sum}}) \end{array} \right\}$$

Here, the predicate variables $Rec_{\texttt{sum}}$ and P respectively represent the recursion relation and an inductive invariant between the argument and the return value of `sum`. The well-foundedness constraint $WF(Rec_{\texttt{sum}})$ requires that the recursion relation $Rec_{\texttt{sum}}$ is well-founded. Thanks to the use of invariant and ranking function synthesis techniques, our Horn

[1] The extension of Horn clause constraints with well-foundedness constraints itself was originally proposed in [30].

constraint solving method presented in Section 6.2 automatically solves the constraint set and obtains an inductive invariant $P(x, y) \equiv y \geq x$, which says whenever a value y is returned it is not less than the argument x, as well as the ranking function $r_{\texttt{sum}}$ as a certificate of the well-foundedness of $Rec_{\texttt{sum}}$.

Our method can automatically verify the termination of more complex functions such as McCarthy's 91 function.

```
let rec mc91 x =
  if x>100 then x-10 else mc91 (mc91 (x+11))
```

From the program, the following constraint set is generated:

$$\left\{\begin{array}{l} P(x), \quad P(x) \Leftarrow P(x-11) \wedge x \leq 111, \\ P(x) \Leftarrow P(n) \wedge Q(n+11, x) \wedge n \leq 100, \\ Q(n, r) \Leftarrow P(n) \wedge r \geq 91 \wedge n = 10 + r, \\ Q(n, r) \Leftarrow Q(n+11, r') \wedge Q(r', r) \wedge P(n) \wedge n \leq 100, \\ Rec_{\texttt{mc91}}(n, n') \Leftarrow P(n) \wedge n \leq 100 \wedge n' = n + 11, \\ Rec_{\texttt{mc91}}(n, n') \Leftarrow P(n) \wedge Q(n+11, n') \wedge n \leq 100, \\ WF(Rec_{\texttt{mc91}}) \end{array}\right\}$$

Here, the predicate variables $Rec_{\texttt{mc91}}$, P, and Q respectively represent the recursion relation, the pre- and post-conditions of `mc91`. Our method automatically solves the constraint set and obtains a tricky ranking function $r_{\texttt{mc91}}(x) = 111 - x$ and an inductive invariant of the function `mc91` in the form of the refinement type

$$(x : \texttt{int}) \rightarrow \left\{ y : \texttt{int} \;\middle|\; \begin{array}{l} y \geq 91 \wedge y \geq x - 10 \wedge \\ (y = 91 \vee y = x - 10) \end{array} \right\}$$

that is strong enough to show $Rec_{\texttt{mc91}} \subseteq \mathit{wfrel}(r_{\texttt{mc91}})$. Note here that the termination verification itself involves invariant synthesis as shown by this example.

Recursive functions on algebraic data structures Let us consider the following list manipulating program `merge` obtained from the standard library of the OCaml language.

```
let rec merge cmp l1 l2 = match l1, l2 with
  | [], l2 -> l2 | l1, [] -> l1
  | h1 :: t1, h2 :: t2 ->
    if cmp h1 h2 <= 0 then h1::merge cmp t1 l2
                      else h2::merge cmp l1 t2
```

The constraint set generated from the program is over not only integers but also lists. Such constraints, however, cannot be handled by the previous Horn constraint based invariant and ranking function synthesis methods. We therefore propose novel techniques to lift the previous methods for solving Horn clause and well-foundedness constraints over numerical values to those for solving constraints over higher-order functions and algebraic data structures: we introduce and use *size functions* for abstracting terms of ADTs in the constraint set into integer terms, in order to obtain an ADT-free over-approximated constraint set that can be handled by the previous methods. For `merge`, we may use the size function

$$\mathit{size}^{\texttt{list}}([\,]) = 1 \qquad \mathit{size}^{\texttt{list}}(h :: t) = 1 + \mathit{size}^{\texttt{list}}(t)$$

on lists that returns the syntactic size of the argument. Our method then automatically verifies the termination of the program by synthesizing a ranking function $r_{\texttt{merge}}(\mathit{cmp}, l_1, l_2) = \mathit{size}^{\texttt{list}}(l_1) + \mathit{size}^{\texttt{list}}(l_2)$ for `merge`. As shown in Section 7, in experiments on the OCaml List and Map modules that manipulate algebraic data structures, the termination of all the functions of the modules was verified by inferring ranking functions on the syntactic sizes of the data structures, though our method can automatically synthesize other size functions if necessary (see below for examples and Section 6 for technical details).

Higher-order functions We need a further twist to precisely handle higher-order functions. Let us consider the following higher-order terminating program.

```
let k1 g () = g () - 1
let k2 n () = n
let rec f g () = if g()<=0 then () else f(k1 g)()
let main n = f (k2 n) ()
```

The termination verification of the function `f` requires a ranking function over the function argument `g` because each recursive call to `f` updates the function argument `g`: the initial closure `k2 n` passed as the argument `g` at `main` returns the integer `n` when applied to $()$ and each recursive call to `f` updates `g` to $\texttt{g}' = \texttt{k1 g}$ such that $\texttt{g}'\,() = \texttt{g}\,() - 1$. Therefore, `g` $()$ eventually returns a non-positive integer and `f` terminates. So that we can apply existing numerical ranking function synthesis techniques to termination verification of such higher-order functions, we introduce an ADT that represents closures possibly passed to the argument. For the above example, we obtain the ADT for the function argument `g`:

```
type cls = K1 of cls | K2 of int
```

Because `g` has the ordinary ML type `unit` $\rightarrow$ `int` and any closure of this type is of the form $\texttt{k1}^m\ (\texttt{k2}\ n)$ for some integers n and $m \geq 0$, the ADT is defined as above using the constructors `K1` and `K2` corresponding to `k1` and `k2`. We then consider the following program obtained from the previous one by inserting a *ghost* parameter (indicated by the square brackets) `cg` of `f` representing the algebraic data structure encoding the closures passed as the function argument `g`:[2]

```
let k1 g () = g () - 1
let k2 n () = n
let rec f [cg] g () =
  if g () <= 0 then () else f [K1 cg] (k1 g) ()
let main n = f [K2 n] (k2 n) ()
```

[2] Our refinement type system formalized in Sections 4 and 5 actually does not introduce such ghost parameters in the program-level but instead represents them in the type-level using function refinement types.

By using the size function for the ADT defined by

$$s(\texttt{K1}\ c) = s(c) - 1 \qquad s(\texttt{K2}\ n) = n,$$

we obtain the ranking function $r_{\texttt{f}}(cg, ()) = s(cg)$ witnessing the termination of the higher-order program. Our implementation reported in Section 7 automatically synthesized the size and ranking functions.

For another example of higher-order programs, consider the following terminating one.

```
let app f x = f x
let rec g x = if x>0 then app g (x-1) else ()
```

Note that g is recursively called indirectly via the higher-order function app. To verify the termination of the function g, we need to precisely know what values are passed indirectly to g by app. Note that the evaluation of app g $(x-1)$ only passes $x-1$ to g and therefore the evaluation of g n always terminates for any n. However, as pointed out by Unno et al. [42], ordinary refinement type systems underlying automated verification methods cannot express that property of app unless a ghost (in their method integer) parameter is inserted before the function argument f of the higher-order function app. To remedy the limitation of the ordinary refinement type systems, we generalize Unno et al.'s approach: we introduce the ADT

```
type cls = Grec of int | G | App of cls
```

that encodes the closures passed as the function argument f to app, in a similar manner to the previous example, and insert a ghost ADT parameter cf before the function argument f of app to obtain

```
let app [cf] f x = f x
let rec g x =
  if x>0 then app [Grec x] g (x-1) else ()
```

Here, both of the constructors Grec and G correspond to g but the former is used in the definition of g and the latter is used outside the definition. Note also that the algebraic data structure Grec x encoding the recursive occurrence of g in the definition of g records the argument x passed to g. By using the size function

$$s(\texttt{Grec}\ x) = x \qquad s(\texttt{G}) = 0 \qquad s(\texttt{App}\ c) = s(c)$$

for the ADT, we obtain the ranking function $r_{\texttt{g}}(x) = x$ and the refinement type

$$(cf : \texttt{cls}) \to (\{x : \texttt{int} \mid 0 \le x < s(cf)\} \to \texttt{unit}) \to \{x : \texttt{int} \mid 0 \le x < s(cf)\} \to \texttt{unit}$$

for app that witness the termination of the higher-order program. Our implementation automatically synthesized the above refinement type as well as the size and ranking functions.

$$E[op(\widetilde{n})] \longrightarrow E[[\![op]\!](\widetilde{n})] \qquad \text{(E-OP)}$$

$$\frac{|\widetilde{x}| = |\widetilde{v}|}{E[(\texttt{fun}^{\ell} f\ \widetilde{x}.\ e)\ \widetilde{v}] \longrightarrow E[[\texttt{fun}^{\ell} f\ \widetilde{x}.\ e/f, \widetilde{v}/\widetilde{x}]e]} \qquad \text{(E-APP)}$$

$$E[\texttt{case}\ C_j(\widetilde{v_j})\ \texttt{of}\ \{C_i(\widetilde{x_i}) \to e_i\}_{i=1}^{m}] \longrightarrow E[[\widetilde{v_j}/\widetilde{x_j}]e_j] \qquad \text{(E-CASE)}$$

$$E[\texttt{let}\ x = v\ \texttt{in}\ e] \longrightarrow E[[v/x]e] \qquad \text{(E-LET)}$$

Figure 1. The operational semantics of the language $\mathcal{L}$

3. The Target Language $\mathcal{L}$

This section presents an ML-like strict higher-order functional language with algebraic data types (ADTs) as the target of our verification method. The syntax is defined by:

$$\begin{array}{rrll}
\text{(expressions)} & e & ::= & x \mid n \mid op(e_1, \ldots, e_{\mathrm{ar}(op)}) \\
& & \mid & \texttt{fun}^{\ell} f\ \widetilde{x}.\ e \mid e_1\ e_2 \\
& & \mid & C(e_1, \ldots, e_{\mathrm{ar}(C)}) \\
& & \mid & \texttt{case}\ e\ \texttt{of}\ \{C_i(\widetilde{x}_i) \to e_i\}_{i=1}^{m} \\
& & \mid & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\
\text{(values)} & v & ::= & n \mid (\texttt{fun}^{\ell} f\ \widetilde{x}.\ e)\ \widetilde{v} \mid C\ (\widetilde{v}) \\
\text{(eval. contexts)} & E & ::= & [\,] \mid E\ e \mid v\ E \mid C(\widetilde{v}, E, \widetilde{e}) \\
& & \mid & \texttt{case}\ E\ \texttt{of}\ \{C_i(\widetilde{x}_i) \to e_i\}_{i=1}^{m} \\
& & \mid & \texttt{let}\ x\ =\ E\ \texttt{in}\ e \\
\text{(ML types)} & A & ::= & \texttt{int} \mid A_1 \to A_2 \mid D
\end{array}$$

Here, meta-variables x and f range over variables. n and op respectively represent integer constants and operations. op includes, for example, $+$, $<$, and $=$. D and C respectively represent ADTs and their constructors. We assume that D includes the unit type unit, the Boolean type bool, the list type list, and user-defined ADTs. Accordingly, C includes the unit value $()$, the Boolean values true and false, the list constructors $[\,]$ and $(::)$, and those for user-defined ADTs. We write $\mathrm{ar}(op)$ and $\mathrm{ar}(C)$ respectively for the arity of op and C. We write $\widetilde{x}$ (resp. $\widetilde{v}$) for a sequence of variables (resp. values). The length of the sequence $\widetilde{x}$ is denoted by $|\widetilde{x}|$. An expression $\texttt{fun}^{\ell} f\ \widetilde{x}.\ e$ represents a recursive function annotated with a unique label ℓ, where $|\widetilde{x}| \ge 1$ and f representing the function itself may occur recursively in the body e. We define $\mathrm{ar}(\ell)$ to be $|\widetilde{x}|$. We write $fvs(e)$ for the set of free variables occurring in e. We assume that expressions are well-typed under the ordinary ML type system $\vdash$. We also assume that the ML type of the constructors for user-defined ADTs is first-order. This restriction is essential because without this, $\texttt{fun}^{\ell} f\ \widetilde{x}.\ e$ may call itself recursively via a function argument in $\widetilde{x}$ even if f does not occur in e. In the definition of values, $(\texttt{fun}^{\ell} f\ \widetilde{x}.\ e)\ \widetilde{v}$ represents a function closure that satisfies $|\widetilde{v}| < |\widetilde{x}|$. The call-by-value operational semantics of the language is shown in Figure 1. There, $\longrightarrow$ is a one-step evaluation relation, which is deterministic, and $[\![op]\!]$ represents the integer function denoted by op. We often write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

4. Refinement Type System $\vdash_P$ for Partial Correctness Verification

We now present our refinement type system $\vdash_P$ for partial correctness verification. The syntax of types is defined by:

$$
\begin{array}{rrl}
\text{(refinement types)} & \tau ::= & \{x : T \mid \phi\} \\
& T ::= & \mathtt{int} \mid (x : \tau_1) \to \tau_2 \mid D \\
\text{(type environments)} & \Gamma ::= & x_1 : \tau_1, \ldots, x_m : \tau_m \\
\text{(formulas)} & \phi ::= & t_1 = t_2 \mid t_1 \leq t_2 \mid \top \mid \bot \mid \neg\phi \\
& \mid & \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \Rightarrow \phi_2 \\
\text{(terms)} & t ::= & x \mid n \mid t_1 + t_2 \mid n \cdot t \\
& \mid & C(t_1, \ldots, t_{\mathrm{ar}(C)}) \mid s(t) \\
\text{(predicates)} & p ::= & \lambda\widetilde{x}.\phi
\end{array}
$$

A refinement type $\{x : T \mid \phi\}$ equipped with a refinement formula ϕ represents the type of the values x of the type T that satisfies ϕ. The scope of x is within ϕ and T. We often abbreviate $\{x : T \mid \top\}$ as T. Note that T can be a function type unlike in ordinary refinement type systems. $(x : \tau_1) \to \tau_2$ is the type of functions that take an argument x of the type τ_1 and return a value of the type τ_2, where the scope of x is within τ_2. For example, $(x : \mathtt{int}) \to \{y : \mathtt{int} \mid y \geq x\}$ is the type of functions whose return value y is not less than the argument x. We abbreviate $(x : \tau_1) \to \tau_2$ as $\tau_1 \to \tau_2$ if x does not occur in τ_2. We write $\mathit{fvs}(\tau)$ for the set of free variables that occur in τ. We define $\mathrm{dom}(\Gamma) = \{x \mid x : \tau \in \Gamma\}$ and $\Gamma(x) = \tau$ if $x : \tau \in \Gamma$.

In this paper, we adopt refinement formulas in the quantifier-free theory of linear integer arithmetic (QFLIA) and equality on the terms of ADTs with *size functions* (ranged over by the meta-variable s) that map values of some ADT D to integers. For example, the size function $\mathtt{len}$ on lists that returns the length is defined by $\mathtt{len}([\,]) = 0$ and $\mathtt{len}(x :: l) = 1 + \mathtt{len}(l)$. We assume that size functions on an ADT D are catamorphisms on D (see Section 6.2.1 for more details). The formulas $\top$ and $\bot$ respectively represent the tautology and the contradiction. We call a predicate p *well-founded* and write $\mathit{WF}(p)$, if the arity of p is $2 \times n$ for some n and there is no infinite sequence $\widetilde{t_1}, \widetilde{t_2}, \ldots$ such that $|\widetilde{t_i}| = n$ and $p(\widetilde{t_i}, \widetilde{t_{i+1}})$ holds for all $i \geq 1$.

The typing rules for the system $\vdash_P$ are shown in Figure 2. A type judgment $\Gamma \vdash_P e : \tau$ means that an expression e has a refinement type τ under a refinement type environment Γ. A subtype judgment $\Gamma \vdash_P \tau_1 <: \tau_2$ indicates that τ_1 is a subtype of τ_2 under Γ. $\mathrm{type}(op)$ in the rule P-Op represents the refinement type of op that abstracts the behavior of the function $[\![op]\!]$ soundly. For example, $\mathrm{type}(+) = (x : \mathtt{int}) \to (y : \mathtt{int}) \to \{z : \mathtt{int} \mid z = x + y\}$. $\mathrm{type}(C)$ in the rule P-Con represents the refinement type of C. By default, $\mathrm{type}(C)$ is defined by $(\widetilde{x} : \widetilde{A}) \to \{x : D \mid x = C(\widetilde{x})\}$ if the ordinary ML type of C is $\widetilde{A} \to D$. Our implementation reported in Section 7 also allows users to refine the argument types $\widetilde{A}$ with pre-conditions (see an explanation of experiments on the OCaml Map module in Section 7 for an example). All the rules except P-Fun are essentially the same as

$$\frac{\Gamma(x) = \{\nu : T \mid \phi\}}{\Gamma \vdash_P x : \{\nu : T \mid \nu = x\}} \quad \text{(P-Var)}$$

$$\frac{\Gamma, \widetilde{x} : \widetilde{\tau}, f : \mathit{PLift}\,(\widetilde{x} : \widetilde{\tau}; \tau; \ell; \epsilon) \vdash_P e : \tau}{\Gamma \vdash_P \mathtt{fun}^{\ell} f\ \widetilde{x}.\ e : \mathit{PLift}\,(\widetilde{x} : \widetilde{\tau}; \tau; \ell; \epsilon)} \quad \text{(P-Fun)}$$

$$\frac{\Gamma \vdash_P e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_P e_2 : \tau_1}{\Gamma \vdash_P e_1\ e_2 : \tau_2} \quad \text{(P-App)}$$

$$\Gamma \vdash_P n : \{\nu : \mathtt{int} \mid \nu = n\} \quad \text{(P-Int)}$$

$$\frac{\mathrm{type}(op) = (x_1 : \tau_1) \to \cdots \to (x_{\mathrm{ar}(op)} : \tau_{\mathrm{ar}(op)}) \to \tau \qquad \Gamma, e_1 : \tau_1, \ldots, e_{i-1} : \tau_{i-1} \vdash_P e_i : \tau_i \quad (\text{for each } i = 1, \ldots, \mathrm{ar}(op))}{\Gamma \vdash_P op(e_1, \ldots, e_{\mathrm{ar}(op)}) : \tau} \quad \text{(P-Op)}$$

$$\frac{\mathrm{type}(C) = (x_1 : \tau_1) \to \cdots \to (x_{\mathrm{ar}(C)} : \tau_{\mathrm{ar}(C)}) \to \tau \qquad \Gamma, e_1 : \tau_1, \ldots, e_{i-1} : \tau_{i-1} \vdash_P e_i : \tau_i \quad (\text{for each } i = 1, \ldots, \mathrm{ar}(C))}{\Gamma \vdash_P C(e_1, \ldots, e_{\mathrm{ar}(C)}) : \tau} \quad \text{(P-Con)}$$

$$\frac{\Gamma \vdash_P e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash_P e_2 : \tau_2 \qquad x \notin \mathit{fvs}(\tau_2)}{\Gamma \vdash_P \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2} \quad \text{(P-Let)}$$

$$\frac{\Gamma \vdash_P e : \{x : D \mid \phi_1\} \qquad \mathrm{type}(C_i) = (\widetilde{x_i} : \widetilde{\tau_i}) \to \{x : D \mid \phi_2\} \qquad \Gamma, \widetilde{x_i} : \widetilde{\tau_i}, x : \{x : D \mid \phi_1 \wedge \phi_2\} \vdash_P e_i : \tau \qquad \mathit{fvs}(\tau) \cap \{\widetilde{x_i}, x\} = \emptyset \quad (\text{for each } i \in \{1, \ldots, m\})}{\Gamma \vdash_P \mathtt{case}\ e\ \mathtt{of}\ \{C_i(\widetilde{x_i}) \to e_i\}_{i=1}^{m} : \tau} \quad \text{(P-Case)}$$

$$\frac{\Gamma \vdash_P e : \tau' \qquad \Gamma \vdash_P \tau' <: \tau}{\Gamma \vdash_P e : \tau} \quad \text{(P-Sub)}$$

$$\frac{\models [\![\Gamma]\!] \wedge \phi_1 \Rightarrow \phi_2 \qquad x : \mathtt{fresh} \qquad \Gamma, x : \{x : \mathtt{unit} \mid \phi_1\} \vdash T_1 <: T_2}{\Gamma \vdash \{\nu : T_1 \mid \phi_1\} <: \{\nu : T_2 \mid \phi_2\}} \quad \text{(S-Ref)}$$

$$\Gamma \vdash T <: T \quad \text{(S-Base)}$$

$$\frac{\Gamma \vdash \tau_1' <: \tau_1 \qquad \Gamma, \nu : \tau_1' \vdash \tau_2 <: \tau_2'}{\Gamma \vdash (\nu : \tau_1) \to \tau_2 <: (\nu : \tau_1') \to \tau_2'} \quad \text{(S-Fun)}$$

$$\mathit{PLift}\,(x : \tau; \tau'; \ell; \widetilde{y}) = \left\{ g : (x : \tau) \to \tau' \;\middle|\; g = \ell^{(|\widetilde{y}|)}(\widetilde{y}) \right\}$$

$$\mathit{PLift}\,(x : \tau, \widetilde{x} : \widetilde{\tau}; \tau'; \ell; \widetilde{y}) = \left\{ g : (x : \tau) \to \mathit{PLift}\,(\widetilde{x} : \widetilde{\tau}; \tau'; \ell; \widetilde{y}, x) \;\middle|\; g = \ell^{(|\widetilde{y}|)}(\widetilde{y}) \right\}$$

$$[\![x_1 : \{\nu : T_1 \mid \phi_1\}, \ldots, x_n : \{\nu : T_n \mid \phi_n\}]\!] = \bigwedge_{i=1}^{n} [x_i/\nu]\phi_i$$

Figure 2. The derivation rules for $\Gamma \vdash_P e : \tau$ & $\Gamma \vdash \tau <: \tau'$

the ones proposed in previous work [8, 20, 31, 40, 42]. In P-FUN, $PLift\ (\widetilde{x} : \widetilde{\tau}; \tau; \ell; \epsilon)$ plays an essential role for precisely reasoning about higher-order functions. In the definition of $PLift$, $\ell^{(i)}$ are the constructors of the special ADT cls explained in Section 2 for encoding all the possible function closures. For each recursive function with the label ℓ, we introduce constructors $\ell^{(0)}, \ldots, \ell^{(\mathrm{ar}(\ell)-1)}$ of cls, which are used to encode non-recursive occurrences of the function (recall `K1`, `K2`, `G`, and `App` in Section 2). If the function labeled with ℓ has the type $A_1 \to \cdots \to A_{\mathrm{ar}(\ell)} \to A$, the ML type environment for the constructors is defined by $\{\ell^{(i)} : A'_1 \to \cdots \to A'_i \to cls \mid i = 0, \ldots, \mathrm{ar}(\ell) - 1\}$, where $A'_j = cls$ if A_j is a function type and $A'_j = A_j$ otherwise. Note that the types thus defined are first-order.

5. Refinement Type System $\vdash_T$ for Total Correctness Verification

In this section, we modify $\vdash_P$ to obtain a refinement type system $\vdash_T$ for total correctness verification. The derivation rules for $\vdash_T$ are almost the same as the ones for $\vdash_P$ shown in Figure 2 except that P-FUN is replaced by the rule T-FUN:

$$\frac{\begin{array}{c}\models WF(\lambda\widetilde{x}\widetilde{y}.\phi) \qquad fvs(\phi) \subseteq \{\widetilde{x}, \widetilde{y}\} \\ \{\widetilde{x}\} \cap \{\widetilde{y}\} = \emptyset \qquad (\widetilde{x} : \widetilde{\tau}) \to \tau = (\widetilde{y} : \widetilde{\tau}') \to \tau' \\ \Gamma, \widetilde{x} : \widetilde{\tau}, f : TLift\ (\widetilde{y} : \widetilde{\tau}'; \lambda\widetilde{x}\widetilde{y}.\phi; \tau'; \ell; \widetilde{x}) \vdash_T e : \tau\end{array}}{\Gamma \vdash_T \mathtt{fun}^{\ell} f\ \widetilde{x}.\ e : TLift\ (\widetilde{x} : \widetilde{\tau}; \lambda\widetilde{x}\widetilde{y}.\top; \tau; \ell; \widetilde{x})}$$

$$TLift\ (x : \{y : T \mid \phi\}; p; \tau'; \ell; \widetilde{y}) = \left\{ g : (x : \{y : T \mid \phi \wedge p(\widetilde{y}, y)\}) \to \tau' \;\middle|\; g = \ell^{(|\widetilde{y}|)}(\widetilde{y}) \right\}$$

$$TLift\ (x : \tau, \widetilde{x} : \widetilde{\tau}; p; \tau'; \ell; \widetilde{y}) = \left\{ g : (x : \tau) \to TLift\ (\widetilde{x} : \widetilde{\tau}; p; \tau'; \ell; \widetilde{y}, x) \;\middle|\; g = \ell^{(|\widetilde{y}|)}(\widetilde{y}) \right\}$$

Here, $TLift$ plays an essential role for checking the well-foundedness of the recursion relation as well as precisely reasoning about higher-order functions. For total correctness verification, cls is extended as follows. For each recursive function with the label ℓ, we introduce new constructors $\ell^{(\mathrm{ar}(\ell))}, \ldots, \ell^{(2\times\mathrm{ar}(\ell)-1)}$ of cls, which are used to encode recursive occurrences of the function (recall `Grec` in Section 2). If the function with the label ℓ has the ordinary ML type $A_1 \to \cdots \to A_{\mathrm{ar}(\ell)} \to A$, the ML type environment for the new constructors is defined by

$$\left\{ \ell^{(\mathrm{ar}(\ell)+i)} : \begin{array}{l} A'_1 \to \cdots \to A'_{\mathrm{ar}(\ell)} \to \\ A'_1 \to \cdots \to A'_i \to cls \end{array} \;\middle|\; i = 0, \ldots, \mathrm{ar}(\ell) - 1 \right\},$$

where $A'_j = cls$ if A_j is a function type and $A'_j = A_j$ otherwise. The first $\mathrm{ar}(\ell)$ arguments represent the actual arguments passed to the function, and the remaining arguments are those passed toq a recursive call of the function.

We now show properties of the type system $\vdash_T$. To this end, we define the denotational semantics of types.

$$[\![\{x : \mathtt{int} \mid \phi\}]\!]_T = \{e \mid \vdash e : \mathtt{int}, \exists v. e \longrightarrow^* v \wedge \models [[\![v]\!]/x]\phi\}$$

$$[\![\{f : (x : \tau_1) \to \tau_2 \mid \phi\}]\!]_T = \left\{ e \;\middle|\; \begin{array}{l} \vdash e : \lfloor \tau_1 \to \tau_2 \rfloor, \exists v. e \longrightarrow^* v \wedge \models [[\![v]\!]/f]\phi \wedge \\ \forall v_1 \in [\![\tau_1]\!]. e\ v_1 \in [\![[[\![v]\!]/f, [\![v_1]\!]/x]\tau_2]\!]_T \end{array} \right\}$$

$$[\![\{x : D \mid \phi\}]\!]_T = \{e \mid \vdash e : D, \exists v. e \longrightarrow^* v \wedge \models [[\![v]\!]/x]\phi\}$$

$$[\![n]\!] = n, \quad [\![(\mathtt{fun}^{\ell} f\ \widetilde{x}.\ e)\ \widetilde{v}]\!] = \ell^{(|\widetilde{v}|)}([\![\widetilde{v}]\!]), \quad [\![C(\widetilde{v})]\!] = C([\![\widetilde{v}]\!])$$

where $\lfloor \tau \rfloor$ represents the ML type obtained by erasing refinement formulas in the refinement type τ. The denotation $[\![\tau]\!]_T$ of τ intuitively represents the set of non-diverging expressions that satisfy the partial correctness specification represented by τ. $[\![v]\!]$ maps the program value v to the corresponding term in the underlying logic. Note that a function closure $(\mathtt{fun}^{\ell} f\ \widetilde{x}.\ e)\ \widetilde{v}$ is mapped to an ADT term $\ell^{(|\widetilde{v}|)}([\![\widetilde{v}]\!])$.

The progress theorem is proved in a standard manner.

Theorem 1 (Progress). *If $\vdash_T e : \tau$, then either e is a value or $e \longrightarrow e'$ for some e'.*

We can prove the following in a similar manner to [43].

Lemma 1 (Denotaion Typing). *If $\vdash_T e : \tau$ then $e \in [\![\tau]\!]_T$.*

The substitution lemma and the type preservation theorem are obtained by Lemma 1.

Theorem 2 (Preservation). *If $\vdash_T e : \tau \wedge e \longrightarrow e'$, $\vdash_T e' : \tau$.*

The soundness of $\vdash_T$ is obtained as a corollary of Lemma 1.

Corollary 1 (Soundness). *If $\vdash_T e : \tau$, then e cannot diverge.*

6. Type Inference Method for $\vdash_T$

We propose a refinement type inference method for the system $\vdash_T$ of total correctness verification. Following the previous approach, we reduce refinement type inference into constraint solving [21, 40]. The constraints generated from a given program in our setting are represented by Horn clauses over integers and ADTs. In addition, we handle well-foundedness constraints to ensure the termination of the given program.

So that we can reuse existing numerical invariant and ranking function synthesis techniques to solve such constraints, we propose a technique called *size abstraction* for abstracting terms of ADTs into integer terms using *size functions*, which are catamorphisms over ADTs. We also propose a counterexample guided method for automatically synthesizing size functions if necessary.

6.1 Constraint generation

This section formalize our Horn constraint solving problems and explains our reduction from refinement type inference to constraint solving based on the previous methods [21, 40].

Horn constraint solving problems A *Horn clause Constraint Set (HCS)* $\mathcal{H}$ is a finite set $\{hc_1, \ldots, hc_m\}$ of Horn clauses. A *Horn clause* hc is defined to be $h \Leftarrow b$ where

- a *head* h is of the form either $P(\widetilde{t})$ or ϕ, and
- a *body* b is of the form $P_1(\widetilde{t}_1) \wedge \ldots \wedge P_m(\widetilde{t}_m) \wedge \phi$.

Recall that ϕ represents a formula over ADTs and integers in this paper. A Horn clause with the head of the form $P(\widetilde{t})$ (resp. ϕ) is called a *definite* clause (resp. a *goal* clause). We abbreviate a Horn clause $h \Leftarrow \top$ as h. We write $fvs(hc)$ for the set of free variables that occur in hc. A *Horn clause and Well-foundedness Constraint Set (HWCS)* $\mathcal{W}$ is the union of an HCS $\mathcal{H}$ and a well-foundedness constraint set $\{WF(P_1), \ldots, WF(P_m)\}$. The set of predicate variables that occur in an HWCS $\mathcal{W}$ is denoted by $pvs(\mathcal{W})$. We say that $\mathcal{W}$ is *ADT-free* if all terms occurring in $\mathcal{W}$ are *not* of an ADT. The *dependency relation* $\lhd_{\mathcal{W}}$ is defined to be the relation that, for all $P, Q \in pvs(\mathcal{W})$, $Q \lhd_{\mathcal{W}} P$ if and only if $Q(\widetilde{t}_1) \Leftarrow \cdots \wedge P(\widetilde{t}_2) \wedge \cdots \in \mathcal{W}$. We write $\lhd^+_{\mathcal{W}}$ for the transitive closure of $\lhd_{\mathcal{W}}$. We say that P is *recursive* if $P \lhd^+_{\mathcal{W}} P$. We call $\mathcal{W}$ *recursion-free* if $pvs(\mathcal{W})$ contains no recursive predicate variable. A *predicate substitution* θ for $\mathcal{W}$ is a map from each predicate variable $P \in pvs(\mathcal{W})$ to a closed predicate $\lambda x_1, \ldots, x_{\mathrm{ar}(P)}.\phi$. We write $\theta\mathcal{W}$ to denote the application of a predicate substitution θ to $\mathcal{W}$. We write $\mathrm{dom}(\theta)$ to represent the domain of θ. We call a predicate substitution θ a *solution* of $\mathcal{W}$, if $\models \theta hc$ for each $hc \in \mathcal{W}$ and θP is well-founded for each P such that $WF(P) \in \mathcal{W}$. An *HWCS solving problem* is the problem of finding a solution for a given $\mathcal{W}$. A *Horn clause and Well-foundedness Constraint Set with Existentially quantified parameters (EHWCS)* is defined to be $\exists\widetilde{x}.\mathcal{W}$, where the parameters $\widetilde{x}$ are shared by the Horn clauses in $\mathcal{W}$. For each $hc \in \mathcal{W}$, the free variables in $fvs(hc) \setminus \{\widetilde{x}\}$ are universally quantified implicitly. We call a pair (θ, σ) of predicate and term substitutions a *solution* of $\exists\widetilde{x}.\mathcal{W}$, if $\mathrm{dom}(\sigma) = \{\widetilde{x}\}$ and θ is a solution of $\sigma\mathcal{W}$. An *EHWCS solving problem* is the problem of finding a solution for a given $\exists\widetilde{x}.\mathcal{W}$. A *Horn clause Constraint Set with Existentially quantified parameters (EHCS)* is similarly defined to be $\exists\widetilde{x}.\mathcal{H}$.

Reduction from refinement type inference Our method reduces a refinement type inference problem into an HWCS solving problem in a similar manner to an existing refinement type inference method [40]. Given a program e, our method first prepares a refinement type template τ_ℓ and a predicate variable Rec_ℓ for each function $\mathtt{fun}^\ell f\ \widetilde{x}.\ e'$ that occur in e. The type template τ contains predicate variables that represent unknown predicates for refinement, and the predicate variable Rec_ℓ represents the unknown recursion relation of the function. Our method then generates an HWCS by type-checking e using the type templates and collecting Horn clause constraints of the form $[\![\Gamma]\!] \wedge \phi_1 \Rightarrow \phi_2$ from the applications of the rule S-REF and well-foundedness constraints of the form $WF(Rec_\ell)$ from the applications of the rule T-REF. We write $Gen(e)$ for the HWCS thus generated from e. We can show the soundness of the reduction in the same way as in [40].

Theorem 3 (Soundness of Reduction)**.** *Let e be a program. If $Gen(e)$ has a solution, there is τ such that $\vdash_T e : \tau$ holds.*

Example 1. *Consider the following functions on lists:*

```
let rec combine l1 l2 = match (l1, l2) with
  | [], [] -> []
  | (a1::l1), (a2::l2) -> (a1,a2)::combine l1 l2
  | _, _ -> assert false
let main l = combine l l
```

We then obtain the refinement type templates $(l : \alpha\ \mathtt{list}) \to (\alpha * \alpha)\ \mathtt{list}$ *for* `main` *and* $(l_1 : \beta\ \mathtt{list}) \to (l_2 : \{l_2 : \gamma\ \mathtt{list} \mid P(l_1, l_2)\}) \to \{r : (\beta * \gamma)\ \mathtt{list} \mid Q(l_1, l_2, r)\}$ *for* `combine` *with predicate variables P and Q which respectively represent unknown pre- and post-conditions of* `combine` *to be inferred.*[3] *We then obtain the following HWCS $\mathcal{W}_{\mathtt{combine}}$ (after simplification):*

$$\left\{\begin{array}{l} 1 : Q([\,], [\,], [\,]) \Leftarrow P([\,], [\,]), \\ 2 : Q(x_1 :: l_1, x_2 :: l_2, (x_1, x_2) :: r) \\ \qquad \Leftarrow P(x_1 :: l_1, x_2 :: l_2) \wedge Q(l_1, l_2, r), \\ 3 : P(l_1, l_2) \Leftarrow P(x_1 :: l_1, x_2 :: l_2), \\ 4 : \bot \Leftarrow P(l_1, l_2) \wedge \left(\begin{array}{l} l_1 = [\,] \wedge l_2 = x_2 :: l_2' \vee \\ l_1 = x_1 :: l_1' \wedge l_2 = [\,] \end{array}\right), \\ 5 : Rec_{\mathtt{comb}}(x_1 :: l_1, x_2 :: l_2, l_1, l_2) \Leftarrow P(x_1 :: l_1, x_2 :: l_2), \\ 6 : WF(Rec_{\mathtt{comb}}), \\ 7 : P(l, l) \end{array}\right\}$$

Here, the labels 1–7 are not a part of the constraint set. $Rec_{\mathtt{comb}}$ represents the recursion relation of `combine`*. The constraint 1 is obtained from the first branch of the* `match`*-expression in the definition of* `combine`*. The constraints 2 and 3 are from the second branch, and the constraint 4 is from the third branch and requires that this branch is unreachable. The constraint 5 defines the recursion relation of* `combine` *and the constraint 6 requires its well-foundedness. The constraint 7 is from the definition of* `main`*.* □

6.2 Constraint solving

In this section, we describe our constraint solving method for HWCSs. The overall structure of the method is shown in Figure 3. Our method takes an HWCS $\mathcal{W}$ over integers and ADTs. In order to exploit existing techniques for solving a recursion- and ADT-free EHCS, our method reduces the original HWCS solving problem into a recursion- and ADT-free EHCS solving problem (Steps 1-3 in Figure 3). The reduction needs to remove (1) ADTs, (2) well-foundedness constraints, and (3) recursive predicate variables from the original constraint set $\mathcal{W}$. To this end, we use (1) size abstraction $SAbs$ (Step 1 in Figure 3, see Section 6.2.1 for details), (2) well-foundedness constraint elimination $Elim$ (Step 2, Section 6.2.2), and (3) Horn clause

[3] For simplicity, predicate variables for refining function types are omitted.

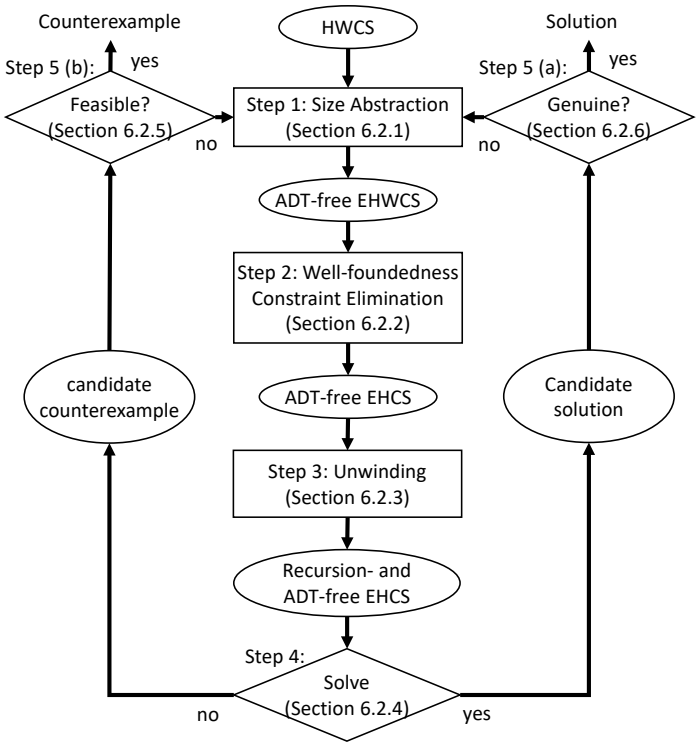

Figure 3. Overall structure of our constraint solving method

unwinding $Unwind$ (Step 3, Section 6.2.3), respectively. $SAbs$ takes an HWCS and a map $\mathcal{T}_{size}$ from ADTs to sequences of (possibly template) size functions, and returns an ADT-free EHWCS. $Elim$ takes an EHWCS and a map $\mathcal{T}_{rank}$ from predicate variables to sequences of (possibly template) linear ranking functions, and returns an EHCS. $Unwind$ takes an EHCS and an integer $k \geq 0$ that represents the degree of constraint unwinding, and returns a recursion-free EHCS. With the three abstractions, we obtain a recursion- and ADT-free EHCS $\mathcal{A}(\mathcal{W}, \mathcal{T}_{size}, \mathcal{T}_{rank}, k) = Unwind(Elim(SAbs(\mathcal{W}, \mathcal{T}_{size}), \mathcal{T}_{rank}), k)$.

We then use existing techniques to solve the reduced constraint set $\mathcal{A}(\mathcal{W}, \mathcal{T}_{size}, \mathcal{T}_{rank}, k)$ (Step 4, Section 6.2.4).

- If a solution (θ, σ) of the reduced constraint set is found, we construct a set of candidate solutions for the original constraint set $\mathcal{W}$ using (θ, σ) as a hint. We then check whether there is a genuine solution for $\mathcal{W}$ in the set (Step 5(a), Section 6.2.5).
  - If it is the case, our method returns it as a solution of $\mathcal{W}$, from which our method constructs refinement types, size functions, and ranking functions of the original program.
  - Otherwise, it is the case that the parameter k for constraint unwinding is not sufficient to obtain a precise enough abstraction. We therefore refine the abstraction by incrementing the parameter k, and repeat the entire process from Step 1 to obtain new candidates.
- Otherwise, we obtain a counterexample witnessing the unsolvability of the reduced constraint set $\mathcal{A}(\mathcal{W}, \mathcal{T}_{size}, \mathcal{T}_{rank}, k)$. The counterexample is a (preferably minimal) unsolvable subset $\exists\widetilde{x}.\mathcal{H}_{cex}$ of $\mathcal{A}(\mathcal{W}, \mathcal{T}_{size}, \mathcal{T}_{rank}, k)$ containing *just one goal clause*. The unsolvability of the reduced $\mathcal{A}(\mathcal{W}, \mathcal{T}_{size}, \mathcal{T}_{rank}, k)$, however, does not imply that of the original $\mathcal{W}$. We therefore construct a candidate counterexample for $\mathcal{W}$ corresponding to $\exists\widetilde{x}.\mathcal{H}_{cex}$, and check its feasibility.
  - If the candidate is feasible, our method returns it as a counterexample witnessing the unsolvability of $\mathcal{W}$.
  - Otherwise, it is the case that either the parameter $\mathcal{T}_{size}$ for size abstraction or the parameter $\mathcal{T}_{rank}$ for well-foundedness constraint elimination is not sufficient to obtain a precise enough abstraction. We therefore refine the abstraction by updating either $\mathcal{T}_{size}$ or $\mathcal{T}_{rank}$ depending on the main cause of the infeasibility, and repeat the entire process from Step 1.

6.2.1 Size abstraction

This section explains $SAbs$ that takes an HWCS $\mathcal{W}$ and a map $\mathcal{T}_{size}$ from ADTs to sequences of (possibly template) size functions, and returns an ADT-free EHWCS.[4] The abstracted constraint set $SAbs(\mathcal{W}, \mathcal{T}_{size})$ has a solution only if the original $\mathcal{W}$ has one. We assume that a size function s for an ADT D with the constructors $\{C_1, \ldots, C_m\}$ is a catamorphism over D of the form $s(x) = \texttt{case}\ x\ \texttt{of}\ \{C_i(\widetilde{x}_i) \to e_i\}_{i=1}^{m}$ where e_i's are linear integer expressions in which recursive calls to s are of the form $s(y)$ for some $y \in \{\widetilde{x}_i\}$. Our implementation reported in Section 7, by default, assume that $\mathcal{T}_{size}(D)$ contains a built-in size function $size^D$ that returns the syntactic size of the given value of D. For example, $size^{\texttt{list}}$ for `list` is defined as $size^{\texttt{list}}(x) = \texttt{case}\ x\ \texttt{of}\ ([\,] \to 1) \mid (h :: t \to 1 + size^{\texttt{list}}(t))$.

We now explain how $SAbs$ transforms a given definite clause $hc \in \mathcal{W}$. (We omit the explanation for goal clauses because they are handled similarly.) For simplicity, we assume that hc is of the form $P_0(\widetilde{x}_0) \Leftarrow P_1(\widetilde{x}_1) \wedge \cdots \wedge P_m(\widetilde{x}_m) \wedge \phi$ where $\widetilde{x}_i$'s are distinct variables and ϕ is the conjunction of atomic formulas such as $t_1 = t_2$, $t_1 \neq t_2$, and $t_1 < t_2$. This involves no loss of generality because we can obtain this normal form using negation and disjunctive normal form transformations. $SAbs$ transforms each $P_i(\widetilde{x}_i)$ by replacing each argument $x \in \{\widetilde{x}_i\}$ of the ADT D into a sequence $\widetilde{x}$ of fresh integer variables that represents $s_1(x), \ldots, s_n(x)$ for $\mathcal{T}_{size}(D) = s_1, \ldots, s_n$. We write $SAbs(s(x))$ to denote the integer variable $y \in \{\widetilde{x}\}$ that represents $s(x)$ and $SAbs^{-1}(y)$ to denote $s(x)$ represented by y. We also write Inv for the set of invariants of the integer variables that represent some $s(x)$. For example, if y repre-

[4] All the existentially quantified parameters of $SAbs(\mathcal{W}, \mathcal{T}_{size})$ are unknown parameters of the template size functions in $\mathcal{T}_{size}$ as well. Therefore, $SAbs$ returns an ADT-free HWCS if $\mathcal{T}_{size}$ contains no template.

sents $size^{\texttt{list}}(x)$, we have $(y > 0) \in Inv$. Our implementation automatically constructs Inv from the definition of size functions using abstract interpretation [6].

The transformation for the ϕ-part is more involved. We first transform ϕ into an equivalent[5] formula $\bigwedge(E_{\mathrm{ADT}} \cup D_{\mathrm{ADT}} \cup I_{\mathrm{INT}})$ where E_{ADT} is a set of ADT equalities of the form $t_1 = t_2$, D_{ADT} is a set of ADT disequalities of the form $t_1 \neq t_2$, and I_{INT} is a set of integer constraints. We here assume that all the integer terms in $E_{\mathrm{ADT}} \cup D_{\mathrm{ADT}}$ are variables. This can be enforced by purification: introduce a fresh integer variable x for each non-variable integer term t in $E_{\mathrm{ADT}} \cup D_{\mathrm{ADT}}$, replace the occurrences of t with x, and add the constraint $x = t$ to I_{INT}.

We then perform size abstraction on $\bigwedge(E_{\mathrm{ADT}} \cup D_{\mathrm{ADT}} \cup I_{\mathrm{INT}})$. Let X be the set of shared integer variables $fvs(E_{\mathrm{ADT}} \cup D_{\mathrm{ADT}}) \cap fvs(I_{\mathrm{INT}})$, Ξ be the set of all the partitions of X, and $\sim_S$ be the equivalence relation on X corresponding to a partition $S \in \Xi$. It then follows that $\bigvee_{S \in \Xi} \bigwedge(E_{\mathrm{ADT}} \cup D_{\mathrm{ADT}} \cup I_{\mathrm{INT}} \cup \{x = y \mid x \sim_S y\} \cup \{x \neq y \mid x \not\sim_S y\})$ is equivalent to $\bigwedge(E_{\mathrm{ADT}} \cup D_{\mathrm{ADT}} \cup I_{\mathrm{INT}})$. We below fix a partition S and explain how to perform size abstraction on each component $C_S = E_{\mathrm{ADT}} \cup D_{\mathrm{ADT}} \cup I_{\mathrm{INT}} \cup \{x = y \mid x \sim_S y\} \cup \{x \neq y \mid x \not\sim_S y\}$. In order to obtain the set E of all the equalities implied by $E_{\mathrm{ADT}} \cup \{x = y \mid x \sim_S y\}$, we compute its congruence closure [27]. We then check whether there is $(t_1 \neq t_2) \in D_{\mathrm{ADT}} \cup \{x \neq y \mid x \not\sim_S y\}$ such that $(t_1 = t_2) \in E$. If it is the case, we replace $\bigwedge C_S$ with $\bot$. Otherwise, we replace $\bigwedge C_S$ with $\bigwedge(SAbs(E, \mathcal{T}_{size}) \cup I_{\mathrm{INT}} \cup \{x \neq y \mid x \not\sim_S y\} \cup Inv)$, where $SAbs(E, \mathcal{T}_{size})$ applies the following $SAbs$ for handling equalities $t_1 = t_2$ in an element-wise fashion to E:

$$SAbs((t_1 = t_2), \mathcal{T}_{size}) = \begin{cases} \bigwedge_{s \in \{\mathcal{T}_{size}(D)\}} ev(s(t_1)) = ev(s(t_2)) & (t_1, t_2 \text{ are } D \text{ terms}) \\ t_1 = t_2 & (t_1, t_2 \text{ are } \texttt{int} \text{ terms}) \end{cases}$$

Here, $ev(s(t))$ represents the result of the partial evaluation of $s(t)$ with each occurrence of remaining $s'(x)$ replaced by $SAbs(s'(x))$.

Example 2. *Recall $\mathcal{W}_{\texttt{combine}}$ in Example 1. Let $\mathcal{T}_1 = \{\texttt{list} \mapsto \epsilon\}$ and $\mathcal{T}_2 = \{\texttt{list} \mapsto size^{\texttt{list}}\}$. We then obtain the following HWCSs by using $\mathcal{T}_1$ and $\mathcal{T}_2$:*

$$SAbs(\mathcal{W}_{\texttt{combine}}, \mathcal{T}_1) = \left\{ \begin{array}{l} 1 : Q() \Leftarrow P(), \\ 2 : Q() \Leftarrow P() \wedge Q(), \\ 3 : P() \Leftarrow P(), \\ 4 : \bot \Leftarrow P(), \\ 5 : Rec_{\texttt{combine}}() \Leftarrow P(), \\ 6 : WF(Rec_{\texttt{combine}}), \\ 7 : P() \end{array} \right\}$$

[5] Strictly speaking, ϕ is equivalent to $\exists \widetilde{x}. \bigwedge(E_{\mathrm{ADT}} \cup D_{\mathrm{ADT}} \cup I_{\mathrm{INT}})$ where $\widetilde{x}$ represents newly introduced fresh variables.

$$SAbs(\mathcal{W}_{\texttt{combine}}, \mathcal{T}_2) = \left\{ \begin{array}{l} 1 : Q(1,1,1) \Leftarrow P(1,1), \\ 2 : Q(1+l_1, 1+l_2, 1+r) \\ \quad \Leftarrow \left(\begin{array}{l} P(1+l_1, 1+l_2) \wedge Q(l_1, l_2, r) \wedge \\ l_1 > 0 \wedge l_2 > 0 \wedge r > 0 \end{array} \right), \\ 3 : P(l_1, l_2) \Leftarrow P(1+l_1, 1+l_2) \wedge l_1 > 0 \wedge l_2 > 0, \\ 4 : \bot \Leftarrow \\ \quad \left(\begin{array}{l} P(l_1, l_2) \wedge l_1 > 0 \wedge l_2 > 0 \wedge l_1' > 0 \wedge l_2' > 0 \wedge \\ (l_1 = 1 \wedge l_2 = 1 + l_2' \vee l_1 = 1 + l_1' \wedge l_2 = 1) \end{array} \right), \\ 5 : Rec_{\texttt{combine}}(1+l_1, 1+l_2, l_1, l_2) \Leftarrow \\ \quad P(1+l_1, 1+l_2) \wedge l_1 > 0 \wedge l_2 > 0, \\ 6 : WF(Rec_{\texttt{combine}}), \\ 7 : P(l, l) \end{array} \right\} \quad \square$$

We can show the following correctness of $SAbs$.

Lemma 2. *Suppose that (θ, σ) is a solution for $SAbs(\mathcal{W}, \mathcal{T}_{size})$. It then follows that $\sigma(SAbs^{-1}(\theta))$ is a solution for $\mathcal{W}$.*

6.2.2 Well-foundedness constraint elimination

This section explains $Elim$ that takes an EHWCS $\exists \widetilde{x}.\mathcal{W}$ and a map $\mathcal{T}_{rank}$ from predicate variables to sequences of (possibly template) linear ranking functions, and returns an EHCS. $Elim$ replaces each well-foundedness constraint $WF(P) \in \mathcal{W}$ with the goal clause $wfrel(\mathcal{T}_{rank}(P))(\widetilde{x}) \Leftarrow P(\widetilde{x})$, where $wfrel(\widetilde{r})$ represents the well-founded relation denoted by the sequence $\widetilde{r}$ of linear ranking functions. Though any ranking template from [25] can be adopted here, in the paper, we henceforth use the following lexicographic ranking template for simplicity:

$$\begin{array}{ll} & wfrel(r_1, \ldots, r_m)(x_1', \ldots, x_n', x_1, \ldots, x_n) \\ & \quad r_1(\widetilde{x}') > r_1(\widetilde{x}) \geq 0 \vee \\ = & \quad r_1(\widetilde{x}') \geq r_1(\widetilde{x}) \wedge r_2(\widetilde{x}') > r_2(\widetilde{x}) \geq 0 \vee \cdots \vee \\ & \quad \left(\bigwedge_{i=1}^{m-1} r_i(\widetilde{x}') \geq r_i(\widetilde{x}) \right) \wedge r_m(\widetilde{x}') > r_m(\widetilde{x}) \geq 0 \end{array}$$

Example 3. *Recall $\mathcal{W}_{\texttt{sum}}$ in Section 2. Let $\mathcal{T}_{rank} = \{Rec_{\texttt{sum}} \mapsto \lambda x.\, c_0 + c_1 x, \lambda x.\, c_2 + c_3 x\}$. We then obtain:*

$$Elim(\mathcal{W}_{\texttt{sum}}, \mathcal{T}_{rank}) = \exists c_0, c_1, c_2, c_3. \left\{ \begin{array}{l} P(x, 0) \Leftarrow x \leq 0, P(x, y) \Leftarrow P(x-1, y-x) \wedge x > 0, \\ Rec_{\texttt{sum}}(x, x') \Leftarrow x' = x - 1 \wedge x > 0, \\ \bot \Leftarrow P(x, y) \wedge x > y, \\ wfrel(\lambda x.\, c_0 + c_1 x, \lambda x.\, c_2 + c_3 x)(x, x') \Leftarrow Rec_{\texttt{sum}}(x, x') \end{array} \right\},$$

where $wfrel(\lambda x.\, c_0 + c_1 x, \lambda x.\, c_2 + c_3 x)(x, x') = c_0 + c_1 x > c_0 + c_1 x' \geq 0 \vee c_0 + c_1 x \geq c_0 + c_1 x' \wedge c_2 + c_3 x > c_2 + c_3 x' \geq 0$. $\square$

We can prove the following correctness of $Elim$.

Lemma 3. *Suppose that (θ, σ) is a solution for $Elim(\exists \widetilde{x}.\mathcal{W}, \mathcal{T}_{rank})$. It then follows that $(\theta, \sigma\restriction_{\{\widetilde{x}\}})$ is a solution for $\exists \widetilde{x}.\mathcal{W}$.*

6.2.3 Constraint unwinding

This section explains $Unwind$ that takes an EHCS $\exists \widetilde{x}.\mathcal{H}$ and an integer $k \geq 0$, and returns the recursion-free EHCS

obtained by unwinding the recursion in $\mathcal{H}$ k-times. Note that, in general, $\mathcal{H}$ with $pvs(\mathcal{H}) = \{P_1, \ldots, P_m\}$ can be expressed as follows:

$$\left\{ \begin{array}{rl} \bot \Leftarrow & F(P_1, \ldots, P_m)(\widetilde{x}_0), \\ P_1(\widetilde{x}_1) \Leftarrow & G_1(P_1, \ldots, P_m)(\widetilde{x}_1), \cdots, \\ P_m(\widetilde{x}_m) \Leftarrow & G_m(P_1, \ldots, P_m)(\widetilde{x}_m) \end{array} \right\},$$

where $F(P_1, \ldots, P_m)$ and $G_i(P_1, \ldots, P_m)$ are of the from:

$$\lambda\widetilde{z}.\exists\widetilde{y}.\left(\begin{array}{l} \big(P_{(1,1)}(\widetilde{t}_{(1,1)}) \wedge \cdots \wedge P_{(1,\ell_1)}(\widetilde{t}_{(1,\ell_1)}) \wedge \phi_1\big) \vee \cdots \vee \\ \big(P_{(n,1)}(\widetilde{t}_{(n,1)}) \wedge \cdots \wedge P_{(n,\ell_1)}(\widetilde{t}_{(n,\ell_1)}) \wedge \phi_n\big) \end{array} \right)$$

Here, $P_{(i,j)} \in \{P_1, \ldots, P_m\}$ and $\exists\widetilde{y}$ binds all free variables in $\widetilde{t}_{(i,j)}$ or ϕ_i except $\widetilde{x}$, $\widetilde{z}$. $Unwind(\exists\widetilde{x}.\mathcal{H}, k)$ is defined by:

$$\exists\widetilde{x}.\left\{ \begin{array}{rl} \bot \Leftarrow & F(P_1^{(0)}, \ldots, P_m^{(0)})(\widetilde{x}_0), \\ P_1^{(0)}(\widetilde{x_1}) \Leftarrow & G_1(P_1^{(1)}, \ldots, P_m^{(1)})(\widetilde{x}_1), \cdots, \\ P_m^{(0)}(\widetilde{x_1}) \Leftarrow & G_m(P_1^{(1)}, \ldots, P_m^{(1)})(\widetilde{x}_m), \\ \vdots & \\ P_1^{(k-1)}(\widetilde{x_1}) \Leftarrow & G_1(P_1^{(k)}, \ldots, P_m^{(k)})(\widetilde{x}_1), \cdots, \\ P_m^{(k-1)}(\widetilde{x_1}) \Leftarrow & G_m(P_1^{(k)}, \ldots, P_m^{(k)})(\widetilde{x}_m) \end{array} \right\},$$

where $P^{(0)}, \ldots, P^{(k)}$ are fresh predicate variables.

Example 4. *Recall $Elim(\mathcal{W}_{\mathtt{sum}}, \mathcal{T}_{rank})$ in Example 3. We obtain the following recursion-free EHCSs:*

$$Unwind(Elim(\mathcal{W}_{\mathtt{sum}}, \mathcal{T}_{rank}), 1) = \exists c_0, c_1, c_2, c_3.$$
$$\left\{ \begin{array}{l} \bot \Leftarrow P^{(0)}(x, y) \wedge x > y, \\ wfrel(\lambda x.c_0 + c_1 x, \lambda x.c_2 + c_3 x)(x, x') \Leftarrow Rec^{(0)}_{\mathtt{sum}}(x, x'), \\ P^{(0)}(x, 0) \Leftarrow x \leq 0, \\ P^{(0)}(x, y) \Leftarrow P^{(1)}(x - 1, y - x) \wedge x > 0, \\ Rec^{(0)}_{\mathtt{sum}}(x, x') \Leftarrow x' = x - 1 \wedge x > 0 \end{array} \right\}$$
$$Unwind(Elim(\mathcal{W}_{\mathtt{sum}}, \mathcal{T}_{rank}), 2) =$$
$$Unwind(Elim(\mathcal{W}_{\mathtt{sum}}, \mathcal{T}_{rank}), 1) \cup$$
$$\left\{ \begin{array}{l} P^{(1)}(x, 0) \Leftarrow x \leq 0, \\ P^{(1)}(x, y) \Leftarrow P^{(2)}(x - 1, y - x) \wedge x > 0, \\ Rec^{(1)}_{\mathtt{sum}}(x, x') \Leftarrow x' = x - 1 \wedge x > 0 \end{array} \right\}$$

□

6.2.4 Constraint solving of rec.- and ADT-free EHCS

We use existing techniques based on Farkas' lemma [2, 15, 42] for solving recursion- and ADT-free EHCS $\exists\widetilde{x}.\mathcal{H}$ reduced from the original constraint set $\mathcal{W}$. If $\exists\widetilde{x}.\mathcal{H}$ is solvable, the techniques allow us to obtain a solution (θ, σ) for $\exists\widetilde{x}.\mathcal{H}$, from which our method constructs candidate solutions for the original $\mathcal{W}$ (see Section 6.2.5 for details). Otherwise, we obtain a counterexample for $\exists\widetilde{x}.\mathcal{H}$ represented by (preferably the smallest) unsolvable subset of $\exists\widetilde{x}.\mathcal{H}$ containing *just one goal clause*, from which we construct a candidate counterexample for $\mathcal{W}$ (see Section 6.2.6 for details).

Example 5. *Recall $Unwind(Elim(\mathcal{W}_{\mathtt{sum}}, \mathcal{T}_{rank}), 1)$ in Example 4. The constraint set is solvable, and an existing recursion- and ADT-free EHCS solver [42] returns a solution (θ, σ) where:*

$$\theta = \left\{ \begin{array}{l} P^{(0)} \mapsto \lambda(x, y).y \geq x, \quad P^{(1)} \mapsto \lambda(x, y).\top, \\ Rec^{(0)}_{\mathtt{sum}} \mapsto \lambda(x, x').x > x' \geq 0, \\ Rec^{(1)}_{\mathtt{sum}} \mapsto \lambda(x, x').\top \end{array} \right\}$$
$$\sigma = \{c_0 = -1, c_1 = 1, c_2 = 0, c_3 = 1\}$$

Note that the candidate ranking functions $\sigma(\mathcal{T}_{rank}(Rec_{\mathtt{sum}})) = (\lambda x.-1 + x, \lambda x.x)$ obtained from the solution of the unwound constraint set already contains a useful clue for the well-foundedness of the recursion relation $Rec_{\mathtt{sum}}$ in $\mathcal{W}$. □

6.2.5 Genuiness checking of candidate solution

This section explains how to check whether a genuine solution for $\mathcal{W}$ exists in a set of candidate solutions obtained from a given solution (θ, σ) for $\mathcal{A}(\mathcal{W}, \mathcal{T}_{size}, \mathcal{T}_{rank}, k)$. We here reuse a previous approach based on predicate abstraction and the least fixed point computation [9].

Because a solution for $\mathcal{W}$ is easily obtained from a solution for $Elim(SAbs(\mathcal{W}, \mathcal{T}_{size}), \mathcal{T}_{rank})$ by using Lemmas 2 and 3, we below consider the set $\Theta(\theta, \sigma)$ of candidate solutions for $Elim(SAbs(\mathcal{W}, \mathcal{T}_{size}), \mathcal{T}_{rank})$ defined by

$$\{(\{P_i \mapsto \bigwedge S_i \mid i = 1, \ldots, m\}, \sigma) \mid \forall i.S_i \subseteq \mathrm{preds}(\theta)(P_i)\}$$

instead of that for $\mathcal{W}$, where $pvs(\mathcal{W}) = \{P_1, \ldots, P_m\}$ and $\mathrm{preds}(\theta)(P)$ represents the set $\{\theta P^{(i)} \mid P^{(i)} \in \mathrm{dom}(\theta)\}$ of predicates associated with P by θ. Note here that the candidate solutions are conjunctions of predicates in $\mathrm{preds}(\theta)(P)$. Our implementation reported in Section 7 is further extended to support candidate solutions that are arbitrary Boolean combinations of predicates: Interested readers are referred to [13, 37].

We now explain how to check whether there is a genuine solution for $Elim(SAbs(\mathcal{W}, \mathcal{T}_{size}), \mathcal{T}_{rank})$ in $\Theta(\theta, \sigma)$ without actually constructing the whole set itself. Here we use the least fixed point computation: starting from the least element (θ_0, σ) with $\theta_0 = \{P_1 \mapsto \bigwedge \mathrm{preds}(\theta)(P_1), \ldots, P_m \mapsto \bigwedge \mathrm{preds}(\theta)(P_m)\}$ of the finite lattice $\Theta(\theta, \sigma)$, we iteratively update θ_i with $\theta_{i+1} = update(\theta_i, \sigma, Elim(SAbs(\mathcal{W}, \mathcal{T}_{size}), \mathcal{T}_{rank}))$ until convergence (i.e., $\theta_i = \theta_{i+1}$ for some i), where $update(\theta, \sigma, \exists\widetilde{x}.\mathcal{H})(P)$ is defined by

$$\bigwedge\{p \in \mathrm{preds}(\theta)(P) \mid \forall\left(P(\widetilde{t}) \Leftarrow b\right) \in \sigma\mathcal{H}. \models p(\widetilde{t}) \Leftarrow \theta(b)\}.$$

Intuitively, $update(\theta, \sigma, \exists\widetilde{x}.\mathcal{H})$ drops conjuncts in $\theta(P)$ that do not satisfy a definite clause in $\exists\widetilde{x}.\mathcal{H}$. Because the lattice is finite, the iterations always converge and the least solution (θ_i, σ) is obtained. We then check whether $\models \theta_i(\sigma hc)$ holds for all goal clauses $hc \in Elim(SAbs(\mathcal{W}, \mathcal{T}_{size}), \mathcal{T}_{rank})$. If so, (θ_i, σ) is a solution for $Elim(SAbs(\mathcal{W}, \mathcal{T}_{size}), \mathcal{T}_{rank})$. Otherwise, there is no genuine solution in the candidate set.

Example 6. *Let us consider the solution (θ, σ) in Example 5 for $Unwind(Elim(\mathcal{H}_{\texttt{sum}}, \mathcal{T}_{rank}), 1)$. We have* $\mathrm{preds}(\theta) = \{P \mapsto \{\lambda(x, y).\, y \geq x\}, Rec_{\texttt{sum}} \mapsto \{\lambda(x, x').\, x > x' \geq 0\}\}$[6]. *We then obtain a genuine solution* $(\mathrm{preds}(\theta), \sigma) \in \Theta(\theta, \sigma)$ *for $Elim(\mathcal{H}_{\texttt{sum}}, \mathcal{T}_{rank})$ as the result of the least fixed point computation. By Lemma 3,* $(\mathrm{preds}(\theta), \sigma\restriction_{\emptyset})$ *is a solution for the original constraint set $\mathcal{H}_{\texttt{sum}}$.* □

6.2.6 Feasibility checking of candidate counterexample

This section explains how to show the unsolvability of the original constraint set $\mathcal{W}$ via the feasibility checking of the candidate counterexample for $\mathcal{W}$ corresponding to the given counterexample $\exists\widetilde{x}.\mathcal{H}_{cex}$ for $\mathcal{A}(\mathcal{W}, \mathcal{T}_{size}, \mathcal{T}_{rank}, k)$. (Recall that $\exists\widetilde{x}.\mathcal{H}_{cex}$ is a unsolvable subset of $\mathcal{A}(\mathcal{W}, \mathcal{T}_{size}, \mathcal{T}_{rank}, k)$ containing just one goal clause.) We also discuss how to refine the parameters $\mathcal{T}_{size}$ and $\mathcal{T}_{rank}$ when the candidate counterexample turned out to be infeasible.

Instead of constructing the candidate counterexample for $\mathcal{W}$ corresponding to $\exists\widetilde{x}.\mathcal{H}_{cex}$ itself, our method constructs the one for $Unwind(Elim(\mathcal{W}, \mathcal{T}_{rank}), k)$, namely, the subset $\exists\widetilde{x}'.\mathcal{H}'_{cex}$ of $Unwind(Elim(\mathcal{W}, \mathcal{T}_{rank}), k)$ such that $SAbs(\exists\widetilde{x}'.\mathcal{H}'_{cex}, \mathcal{T}_{size}) = \exists\widetilde{x}.\mathcal{H}_{cex}$. Our implementation achieves this by tracking the correspondence between Horn clauses before and after the reduction. We then check whether the candidate has a solution.

- If $\exists\widetilde{x}'.\mathcal{H}'_{cex}$ has a solution, it is concluded that the parameter $\mathcal{T}_{size}$ used for size abstraction caused the imprecise abstraction leading to the infeasible candidate. We therefore refine $\mathcal{T}_{size}$ by adding a new template size function for each ADT that occur in $\exists\widetilde{x}'.\mathcal{H}'_{cex}$, and repeat the entire process.
- Otherwise, we have two possibilities:
  1. The unsolvable goal clause $hc \in \exists\widetilde{x}'.\mathcal{H}'_{cex}$ corresponds to $WF(P) \in \mathcal{W}$ (i.e., $hc = Elim(WF(P), \mathcal{T}_{rank})$). It may then be the case that hc was unsolvable because $\mathcal{T}_{rank}$ was insufficient. In this case, we cannot conclude that the candidate counterexample is feasible, and thus conservatively refine $\mathcal{T}_{rank}$ by adding a new template ranking function for P, and repeat the entire process from Step 1.
  2. The unsolvable goal clause $hc \in \exists\widetilde{x}'.\mathcal{H}'_{cex}$ corresponds to a goal clause in $\mathcal{W}$. In this case, the candidate counterexample is feasible and our method returns it as a counterexample for the original $\mathcal{W}$, which thus witnesses the untypability of the original program.

Example 7. *Recall $\mathcal{W}_{\texttt{combine}}$ and $\mathcal{T}_1$ in Example 2. Suppose that we obtain the counterexample $\exists\widetilde{x}.\mathcal{H}_{cex} = \{4 : \bot \Leftarrow P^{(0)}(), 7 : P^{(0)}()\}$ of $Unwind(SAbs(\mathcal{W}_{\texttt{combine}}, \mathcal{T}_1), 1)$. We then obtain the candidate counterexample*

$$\left\{ \begin{array}{l} 4 : \bot \Leftarrow P^{(0)}(l_1, l_2) \wedge \left(\begin{array}{l} l_1 = [\,] \wedge l_2 = x_2 :: l'_2 \vee \\ l_1 = x_1 :: l'_1 \wedge l_2 = [\,] \end{array} \right), \\ 7 : P^{(0)}(l, l) \end{array} \right\}$$

for $\mathcal{W}_{\texttt{combine}}$ corresponding to $\exists\widetilde{x}.\mathcal{H}_{cex}$. Because this constraint set has a solution, our method concludes that the infeasible candidate counterexample is caused by the insufficiency of $\mathcal{T}_{size}$. □

7. Implementation and Experiments

We have implemented a fully-automated type checking and inference tool for total correctness verification of higher-order functional programs with ADTs written in the OCaml language based on the proposed method. We have evaluated our method with widely used OCaml libraries and tricky higher-order programs used as benchmarks in previous studies on termination verification of functional programs [3, 19, 23, 34, 35, 44]

7.1 Implementation

Our tool supports the core of the OCaml language including ADTs and higher-order functions. The tool currently does not support OCaml's imperative features such as reference cells, exceptions, and advanced features like objects.

Our tool takes as input an OCaml program and total correctness specifications expressed as refinement types, and verifies each function whose specification is given. Our tool does not require users to provide invariants of auxiliary functions. Our tool automatically infers them that are sufficient to verify the target functions whose specifications are provided. When the verification succeeds, our tool reports inferred refinement types, ranking functions, and size functions, as a certificate of the total correctness. Our tool is fully automated but also allows users to annotate functions with refinement types, ranking functions, and size functions to aid verification. The annotations can be written using OCaml's language feature called "attributes" supported by OCaml 4.02 or later.

7.2 Experiments on OCaml libraries

We have used the OCaml's List and Map modules as benchmarks for total correctness verification. The List module implements standard list operations and Map module implements association tables using balanced trees. We used our tool to verify that all the module interface functions terminate and do not raise unexpected exceptions such as `Failure` and `Assert_failure`.

Table 2 summarizes the results of the experiments, which covered 67 interface functions totaling 700 non-comment lines of source code. **Mod** is the module name. **LOC** represents the number of non-comment lines of the source code, and **Fun** indicates the number of interface functions. **Size**, **Inv**, and **Rank** respectively represent the annotation number

[6] Useless predicates $\lambda\widetilde{x}.\top$ are omitted here.

Mod	LOC	Fun	Size	Inv	Rank	Time
List	395	44	0	2	0	22.3
Map	305	23	0	0	0	39.5

Table 2. Experiment results on OCaml modules

of size functions, refinement types, and ranking functions we needed for verification. **Time** denotes the total time, in seconds, required to verify all the interface functions. The experiments are conducted on a machine with 1.7GHz Intel Core i7 CPU and 8GB of RAM. We next explain the experiment results for each module.

7.2.1 List Module

The OCaml List module defines standard list manipulating functions. As shown in Table 2, our tool automatically verified all the functions of the module with only two annotations. In the experiments, we provided some of the functions with specifications that have preconditions. For example, consider the function `nth` that takes a list l and an integer n, and returns the $n+1$-th element of the list l. According to the OCaml library manual, this function raises an exception if n is negative or n is not less than the length of l. Here, the length of l is represented as $size^{\texttt{list}}(l) - 1$ using the default size function. We thus provided the refinement type

$$(l : \alpha\ \texttt{list}) \to \left\{n : \texttt{int} \;\middle|\; 0 \leq n < size^{\texttt{list}}(l) - 1\right\} \to \alpha$$

as the specification of `nth` that has a precondition to avoid unexpected exceptions. Our tool then successfully verified the function by automatically inferring a ranking function $r_{\texttt{nth}}(l, n) = size^{\texttt{list}}(l) - 1$ of `nth`, and the type

$$(l : \alpha\ \texttt{list}) \to \left\{n : \texttt{int} \;\middle|\; \begin{array}{l} 0 \leq n < size^{\texttt{list}}(l) - 1 \\ \wedge\ 2 \leq size^{\texttt{list}}(l) \end{array}\right\} \to \alpha$$

of the auxiliary function `nth_aux`. We also specified preconditions for the functions `hd` and `map2` that otherwise raise exceptions as explained in the OCaml library manual.

For exceptions like `Not_found` that are usually expected and handled, instead of specifying preconditions to avoid them, we encoded exception handlers using higher-order functions. For example, consider the following function `find` that may raise the `Not_found` exception:

```
let rec find p = function
  | [] -> raise Not_found
  | x::l -> if p x then x else find p l
```

This is transformed to the one that never raises an exception:

```
type exc = Not_found
let rec find p l ok ex = match l with
  | [] -> ex Not_found
  | x::l -> if p x then ok x else find p l ok ex
```

Here, the function arguments `ok` and `ex` represent continuations that are respectively used when some value is returned by `find` and the exception `Not_found` is raised. This encoding can be automated by a selective CPS transformation [33].

The two annotations we manually provided were for auxiliary functions `chop` and `sort`′ of `sort`:

$$\begin{array}{l} \texttt{chop} : (x : \{x : \texttt{int} \mid ?\}) \to \\ \qquad (l : \left\{l : \alpha\ \texttt{list} \;\middle|\; size^{\texttt{list}}(l) - 1 \geq x\right\}) \to \\ \qquad \left\{r : \alpha\ \texttt{list} \;\middle|\; size^{\texttt{list}}(r) = size^{\texttt{list}}(l) - x\right\} \\ \texttt{sort}' : (\alpha \to \alpha \to \texttt{int}) \to (n : \{n : \texttt{int} \mid n \geq 2\}) \to \\ \qquad \left\{l : \alpha\ \texttt{list} \;\middle|\; n \leq size^{\texttt{list}}(l) - 1\right\} \to \texttt{list} \end{array}$$

Here, $?$ represents an unknown refinement formula which is required to be inferred by our tool. Though these annotations are necessary at present, our tool can benefit from future advances in invariant synthesis techniques to further reduce the annotation burden.

7.2.2 Map Module

The OCaml Map module implements association tables (a.k.a. finite maps or dictionaries) using balanced trees. The module defines the following ADT of balanced trees:

```
type 'a t =
  Empty | Node of 'a t * key * 'a * 'a t * int
```

Our tool automatically generated the default size function $size(x) = \texttt{case}\ x\ \texttt{of}\ (\texttt{Empty} \to 1) \mid (\texttt{Node}(t_1, k, x, t_2, h) \to 1 + size(t_1) + size(t_2))$ for the ADT. The functions of the Map module assumes an invariant of the ADT that the fifth argument of the constructor `Node` contains the height of the tree. The termination verification of the interface functions of this module required a weaker invariant that the height contained in the fifth argument of `Node` is not less than 1. Our tool allows users to specify such invariants as refinement types of the constructors. We thus specified the invariant as the refinement type

$$\alpha\ \texttt{t} \to \texttt{key} \to \alpha \to \alpha\ \texttt{t} \to \{h : \texttt{int} \mid h \geq 1\} \to \alpha\ \texttt{t}$$

of `Node`. With only this, our tool automatically verified all the functions in the Map module. For example, consider the following functions:

```
let bal l x d r = ...
let rec min_binding t = match t with
  | Empty -> assert false
  | Node(Empty, x, d, r, _) -> (x, d)
  | Node(l, x, d, r, _) -> min_binding l
let rec remove_min_binding t = match t with
  | Empty -> assert false
  | Node(Empty, x, d, r, _) -> r
  | Node(l, x, d, r, _) ->
    bal (remove_min_binding l) x d r
let merge t1 t2 = match (t1, t2) with
  | (Empty, t) -> t | (t, Empty) -> t
  | (_, _) -> let (x, d) = min_binding t2 in
              bal t1 x d (remove_min_binding t2)
```

Function	time	Function	time
Ackermann	7.59	churchNum	3.19
alias_partial	0.27	foldr	0.49
append	0.14	indirect	0.58
binomial	0.81	indirectHO	0.50
Fibonacci	0.19	indirectIntro	0.72
loop2	0.61	map	4.59
McCarty91	0.49	quicksort	0.26
zip	0.19	toChurch	0.27
CE-0CFA	0.85	up_down	5.66
CE-1CFA	5.10	x_plus_2^n	0.43
CE-Jones_bohr	3.44		

Table 3. Experiment results on the benchmark set from [23]

The function `merge` merges the two argument trees, the function `min_binding` returns the smallest element of the given tree, and the function `min_binding_remove` removes the smallest element from the given tree. These tree manipulating functions internally calls the auxiliary function `bal` for balancing trees. Our tool automatically verified the total correctness of the function `merge` by inferring the ranking functions $r_{\texttt{min\_binding}}(t) = r_{\texttt{remove\_min\_binding}}(t) = size(t) - 1$ and the types of the auxiliary function with necessary preconditions:

$$\texttt{bal} : \alpha\ \texttt{t} \to \texttt{key} \to \alpha \to \alpha\ \texttt{t} \to \alpha\ \texttt{t}$$
$$\texttt{min\_binding} : \{t : \alpha\ \texttt{t} \mid 3 \leq size(t)\} \to \texttt{key} \times \texttt{int}$$
$$\texttt{remove\_min\_binding} : \{t : \alpha\ \texttt{t} \mid 3 \leq size(t)\} \to \alpha\ \texttt{t}$$

7.3 Tricky recursive functions obtained from [23]

We have tested our tool with the benchmark set provided by Kuwahara et al. [23], which consists of tricky recursive functions obtained from previous studies on termination verification of higher-order functional programs. The results of the experiments are summarized in Table 3. The first 8 functions are first-order functions that require complex ranking functions, and the last 13 functions are higher-order functions that exhibit a complex higher-order control flow. Our tool required a ranking function annotation only for Ackermann. However, for other programs, our tool verified all the programs efficiently without any annotation.

8. Related Work

There are four major approaches to termination verification. This section compares them with our method.

Type-based analysis Sized types [1, 3, 18] and refinement types [43, 44] have been used to verify the termination of recursive functions. They annotate types with size metrics and check that the arguments of recursive calls are with strictly smaller metrics. While our method is fully-automated, [1, 18, 44] are not. [43] uses heuristics to alleviate the annotation burden on users but is not integrated with invariant and ranking function synthesizers. [3] is fully-automated but does not support higher-order functions.

Transition invariants For first-order imperative programs, transition invariants based approach to termination verification has been a great success [4, 5, 29]. These methods reduce transition invariant verification (i.e. binary reachability analysis) into plain reachability analysis. Kuwahara et al. has recently lifted that approach to higher-order functional programs [23]. Their method can handle higher-order functions precisely (actually their method is relatively complete with respect to the soundness and completeness of the backend reachability checker and ranking function synthesizer). Their method, however, does not support ADTs and is not efficient enough to verify real-world programs such as OCaml library modules, due to the program transformation they use, which puts heavy burden on the backend reachability checker.

Size-change analysis Size-change analysis [19, 24, 34–36] for termination verification comprises two steps: first constructs a size-change graph, and then analyzes the graph to decide if the program is terminating by checking every infinite call sequence would cause an infinite descent in some well-founded data values. The methods proposed in [19, 34, 35] can handle higher-order functions. However, these methods often fail to verify the termination of programs with a value-dependent control flow because these methods use a control flow analysis to statically approximate the possible calls that the program would make and does not generate and exploit numerical invariants unlike in our method. Like our method, [19, 34] can prove the well-foundedness of recursion relations on function closures. They, however, use a priori order (namely, the subtree relation), while our method can synthesize and use arbitrary linear size functions.

Term rewriting Similarly to size-change approach to termination, the approach based on term rewriting [10, 11, 16] involves two steps: first transforms the given program into a term rewriting system, and then applies a termination verifier for term rewriting systems to the obtained system. As with the size-change approach, the first step statically approximates (value-dependent) control flow of the original program, which causes the loss of analysis precision.

9. Conclusion

We have proposed a novel refinement type system for total correctness verification of higher-order functional programs with ADTs. We have also presented a fully-automated type inference method based on invariant and ranking function synthesis techniques. The main advantage of the proposed method is that it can automatically infer invariants and ranking functions over not only integers but also higher-order functions and algebraic data structures by synthesizing and using size functions. We have implemented a refinement type checking and inference tool based on the proposed

method, and confirmed that it can verify the total correctness of (1) widely used OCaml libraries with a modest annotation burden in a reasonable time and (2) small but tricky recursive functions that require precise analysis of the higher-order control flow and/or synthesis of complex ranking functions. As a future work, we are planning to extend the proposed refinement type system to verification of general liveness properties. We are also working toward proving the relative completeness of the proposed system with respect to an oracle for validity of refinement formulas as with Hoare logics and a relatively complete refinement type system for partial correctness verification [42].

References

[1] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, Feb. 2004.

[2] T. A. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *CAV '13*, volume 8044 of *LNCS*, pages 869–882. Springer, 2013.

[3] W.-N. Chin and S.-C. Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.

[4] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *SAS '05*, volume 3672 of *LNCS*, pages 87–101. Springer, 2005.

[5] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI '06*, pages 415–426. ACM, 2006.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*, pages 238–252. ACM, 1977.

[7] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[8] C. Flanagan. Hybrid type checking. In *POPL '06*, pages 245–256. ACM, 2006.

[9] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME '01*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[10] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *RTA '04*, volume 3091 of *LNCS*, pages 210–220. Springer, 2004.

[11] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2):7:1–7:39, 2011.

[12] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI '12*, pages 405–416. ACM, 2012.

[13] S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI '09*, volume 5403 of *LNCS*, pages 120–135. Springer, 2009.

[14] A. Gupta, C. Popeea, and A. Rybalchenko. Solving recursion-free horn clauses over LI+UIF. In *APLAS '11*, volume 7078 of *LNCS*, pages 188–203. Springer, 2011.

[15] K. Hashimoto and H. Unno. Refinement type inference via horn constraint optimization. In *SAS '15*, volume 9291 of *LNCS*, pages 199–216. Springer, 2015.

[16] N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205 (4):474–511, 2007.

[17] K. Hoder, N. Bjørner, and L. de Moura. μZ: An efficient engine for fixed points with constraints. In *CAV '11*, volume 6806 of *LNCS*, pages 457–462. Springer, 2011.

[18] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96*, pages 410–423. ACM, 1996.

[19] N. D. Jones and N. Bohr. Call-by-value termination in the untyped lambda-calculus. *Logical Methods in Computer Science*, 4(1), 2008.

[20] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI '09*, pages 304–315. ACM, 2009.

[21] K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *ESOP '07*, volume 4421 of *LNCS*, pages 505–519. Springer, 2007.

[22] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI '11*, pages 222–233. ACM, 2011.

[23] T. Kuwahara, T. Terauchi, H. Unno, and N. Kobayashi. Automatic termination verification for higher-order functional programs. In *ESOP '14*, volume 8410 of *LNCS*, pages 392–411. Springer, 2014.

[24] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL '01*, pages 81–92. ACM, 2001.

[25] J. Leike and M. Heizmann. Ranking templates for linear loops. In *TACAS '14*, volume 8413 of *LNCS*, pages 172–186. Springer, 2014.

[26] K. McMillan and A. Rybalchenko. Computing relational fixed points using interpolation. Technical Report MSR-TR-2013-6, 2013.

[27] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[28] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI '04*, volume 2937 of *LNCS*, pages 239–251. Springer, 2004.

[29] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS '04*, pages 32–41. IEEE, 2004.

[30] C. Popeea and A. Rybalchenko. Compositional termination proofs for multi-threaded programs. In *TACAS '12*, volume 7214 of *LNCS*, pages 237–251. Springer, 2012.

[31] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI '08*, pages 159–169. ACM, 2008.

[32] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV '13*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013.

[33] R. Sato, H. Unno, and N. Kobayashi. Towards a scalable software model checker for higher-order programs. In *PEPM '13*, pages 53–62. ACM, 2013.

[34] D. Sereni. *Termination analysis of higher-order functional programs*. PhD thesis, Magdalen College, 2006.

[35] D. Sereni. Termination analysis and call graph construction for higher-order functional programs. In *ICFP '07*, pages 71–84. ACM, 2007.

[36] D. Sereni and N. D. Jones. Termination analysis of higher-order functional programs. In *APLAS '05*, volume 3780 of *LNCS*, pages 281–297. Springer, 2005.

[37] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI '09*, pages 223–234. ACM, 2009.

[38] T. Terauchi. Dependent types from counterexamples. In *POPL '10*, pages 119–130. ACM, 2010.

[39] T. Terauchi and H. Unno. Relaxed stratification: A new approach to practical complete predicate refinement. In *ESOP '15*, volume 9032 of *LNCS*, pages 610–633. Springer, 2015.

[40] H. Unno and N. Kobayashi. Dependent type inference with interpolants. In *PPDP '09*, pages 277–288. ACM, 2009.

[41] H. Unno and T. Terauchi. Inferring simple solutions to recursion-free horn clauses via sampling. In *TACAS '15*, volume 9035 of *LNCS*, pages 149–163. Springer, 2015.

[42] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In *POPL '13*, pages 75–86. ACM, 2013.

[43] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton Jones. Refinement types for Haskell. In *ICFP '14*, pages 269–282. ACM, 2014.

[44] H. Xi. Dependent types for program termination verification. In *LICS '01*, pages 231–242. IEEE, 2001.

[45] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99*, pages 214–227. ACM, 1999.