Parameterized Recursive Refinement Types for Automated Program Verification

Ryoya Mukai, Naoki Kobayashi(✉), and Ryosuke Sato

The University of Tokyo, Tokyo, Japan
koba@is.s.u-tokyo.ac.jp

Abstract. Refinement types have recently been applied to program verification, where program verification problems are reduced to type checking or inference problems. For fully automated verification of programs with recursive data structures, however, previous refinement type systems have not been satisfactory: they were not expressive enough to state complex properties of data, such as the length and monotonicity of a list, or required explicit declarations of precise types by users. To address the problem above, we introduce *parameterized recursive refinement types* (PRRT), which are recursive datatypes parameterized by integer parameters and refinement predicates; those parameters can be used to express various properties of data structures such as the length/sortedness of a list and the depth/size of a tree. We propose an automated type inference algorithm for PRRT, by a reduction to the satisfiability problem for CHCs (Constrained Horn Clauses). We have implemented a prototype verification tool and evaluated the effectiveness of the proposed method through experiments.

1 Introduction

There has been a lot of progress on automated/semi-automated verification techniques for functional programs, such as those based on higher-order model checking [6,14,16] and refinement types [2,15,17,18,20–22,24,25]. Fully automated verification of functional programs using recursive data structures, however, still remains a challenge. In the present paper, we follow the approach using refinement types, and introduce *parameterized recursive refinement types* and a type inference procedure for them.

Refinement types can be used to express various properties of recursive data types. For example, if we are interested in the length of an integer list, we can prepare a type of the form $\texttt{ilistL}[n]$, which describes a list of *length* n, and assign the following types to constructors:

$$\begin{aligned}&\texttt{Nil} : \texttt{ilistL}[0]\\ &\texttt{Cons} : \forall n.\texttt{int} \times \texttt{ilistL}[n] \to \texttt{ilistL}[n+1]\end{aligned}$$

The type of `Cons` indicates that `Cons` takes a pair consisting of an integer and a list of length n as an argument, and returns a list of length $n + 1$. If we are interested in the sortedness of a list (in the ascending order) instead, we may prepare a type of the form $\texttt{ilistS}[b, x]$, which describes a list consisting of elements no less than x, where the additional Boolean parameter b denotes whether the list is null (thus, if b is true, the value of x should be ignored). The following types can then be assigned to the constructors. (Actually, the second parameter 0 of the type of `Nil` does not matter and may be any other value.)

$$\texttt{Nil} : \texttt{ilistS}[\texttt{true}, 0]$$
$$\texttt{Cons} : \forall b, x, y.\ x : \texttt{int} \times \{\texttt{ilistS}[b, y] \mid \neg b \Rightarrow x \leq y\} \rightarrow \texttt{ilistS}[\texttt{false}, x]$$

Once an appropriate refinement type is assigned to each occurrence of a constructor, a standard procedure for automated/semi-automated refinement type inference (e.g., based on a reduction to the CHC solving problem [2,15,18,25]) is applicable.

A main problem in applying the refinement type approach above to the *fully-automated* verification is that each constructor has more than one refinement type, and it is unclear which type should be used for each occurrence of the constructor (unless a programmer explicitly declares it). For example, for a sorting function `sort`, an input list is a plain, unsorted list, while the output list should be sorted; hence the latter should have type $\texttt{ilistS}[b, x]$ for some b, x. In the context of fully automated verification, we cannot expect a programmer to declare the types like $\texttt{ilistL}[n]$ and $\texttt{ilistS}[b, x]$ above. Thus, an automated verification tool should choose appropriate refinements of recursive data types from infinitely many candidates.

To address the problem above, we parameterize recursive types with integers and predicates, and assign generic types to data type constructors. For example, for integer lists, we prepare a parameterized type $\texttt{ilist}\langle n; e_{\texttt{Nil}}, (\varphi_{\texttt{Cons}}, e_{\texttt{Cons}})\rangle$, where n is an integer denoting the number of integer parameters, $\varphi_{\texttt{Cons}}$ is a predicate on integers, and $e_{\texttt{Nil}}$ and $e_{\texttt{Cons}}$ are functions on integer tuples, and we assign the following types to constructors:

$$\begin{aligned}
&\texttt{Nil} : \forall k, P_{\texttt{Cons}}, f_{\texttt{Nil}}, f_{\texttt{Cons}}.\texttt{ilist}\langle k; f_{\texttt{Nil}}, (P_{\texttt{Cons}}, f_{\texttt{Cons}})\rangle[f_{\texttt{Nil}}()]\\
&\texttt{Cons} : \forall k, P_{\texttt{Cons}}, f_{\texttt{Nil}}, f_{\texttt{Cons}}.\forall \widetilde{y}.\\
&\qquad \{x : \texttt{int} \times \texttt{ilist}\langle k; f_{\texttt{Nil}}, (P_{\texttt{Cons}}, f_{\texttt{Cons}})\rangle[\widetilde{y}] \mid P_{\texttt{Cons}}(x, \widetilde{y})\}\\
&\qquad\qquad \rightarrow \texttt{ilist}\langle k; f_{\texttt{Nil}}, (P_{\texttt{Cons}}, f_{\texttt{Cons}})\rangle[f_{\texttt{Cons}}(x, \widetilde{y})]
\end{aligned}$$

Here, (i) $P_{\texttt{Cons}}$ is a predicate variable, (ii) $f_{\texttt{Nil}}$ and $f_{\texttt{Cons}}$ are functions of types $\texttt{unit} \rightarrow \texttt{int}^k$ and $\texttt{int}^{k+1} \rightarrow \texttt{int}^k$ respectively, and (iii) $\widetilde{y}$ is a sequence of k integer variables (where k is the first parameter of `ilist`). By changing the part $\langle k; f_{\texttt{Nil}}, (P_{\texttt{Cons}}, f_{\texttt{Cons}})\rangle$, we can express various list properties. For example, list type constructors `ilistL` and `ilistS` can be defined as follows:

$$\texttt{ilistL} := \texttt{ilist}\langle 1; \lambda().0, (\lambda(x, y).\texttt{true}, \lambda(x, y).y + 1)\rangle$$
$$\texttt{ilistS} := \texttt{ilist}\langle 2; \lambda().(0, 0), (\lambda(x, y_1, y_2).y_1 > 0 \Rightarrow x \leq y_2, \lambda(x, y_1, y_2).(1, x))\rangle.$$

In fact, by instantiating the parameters $k, P_{\texttt{Cons}}, f_{\texttt{Nil}}$ and $f_{\texttt{Cons}}$ to 1, $\lambda(x,y).\texttt{true}$, $\lambda().0$, and $\lambda(x,y).y+1$ respectively, we obtain the following types for `Nil` and `Cons`:

$$\begin{aligned}&\texttt{Nil} : \texttt{ilistL}[0]\\ &\texttt{Cons} : \forall y.\{x : \texttt{int} \times \texttt{ilistL}[y] \mid \texttt{true}\} \rightarrow \texttt{ilistL}[y+1],\end{aligned}$$

which corresponds to the types of `Nil` and `Cons` given for `ilistL`. Similarly, by instantiating the parameters $k, P_{\texttt{Cons}}, f_{\texttt{Nil}}$ and $f_{\texttt{Cons}}$ to $2, \lambda(x, y_1, y_2).y_1 > 0 \Rightarrow x \leq y_2, \lambda().(0,0)$, and $\lambda(x, y_1, y_2).(1, x)$ respectively, we obtain the types of `Nil` and `Cons` given for `ilistS`.

The remaining question is how to automatically assign an appropriate instantiation of parameterized recursive types to each occurrence of a constructor. To this end, we first pick the values of $k, f_{\texttt{Nil}}, f_{\texttt{Cons}}$ (in the case of lists; we will deal with more general recursive data types in the following sections) in a certain heuristic manner, and prepare a predicate variable for $P_{\texttt{Cons}}$. We can then reduce the problem of refinement type inference to the CHC satisfiability problem [1] in a standard manner [2,18], and use an automated CHC solver [2,4,7]. If the refinement type inference fails, that may be due to the lack of sufficient parameters; thus, we increase the value of k and accordingly update the guess for $f_{\texttt{Nil}}$ and $f_{\texttt{Cons}}$ so that the resulting refinement types are strictly more expressive. This refinement loop may not terminate due to the incompleteness of the type system discussed later in Sect. 3, but we can guarantee a weak form of relative completeness, that if a program is typable, then the type inference procedure terminates eventually under the hypothetical completeness assumption of the underlying CHC solver, as discussed later in Sect. 4.

We have implemented the procedure sketched above, and succeeded in fully automatic verification of several small but challenging programs using lists and trees. Our contributions are summarized as follows.

– The design of parameterized recursive refinement types (PRRT): the idea of parameterizing recursive types with some indices goes back at least to Xi and Pfenning's work [24], and that of parameterization of types with refinement predicates has also been proposed by Vazou et al. [21]. We believe, however, that the specific combination of the parameterizations, specifically designed with fully automated verification in mind, is new.
– An inference procedure for PRRTs, its implementation and experiments.

The rest of this paper is structured as follows. Section 2 introduces the target language of our verification method based on parameterized recursive refinement types. Section 3 proposes a new refinement type system, and Sect. 4 explains a type inference procedure, which serves as a program verification procedure. Section 5 reports an implementation and experimental results. Section 6 discusses related work, and Sect. 7 concludes the paper.

2 Target Language

We consider a first-order[1] call-by-value functional language as the target of our refinement type inference.

2.1 Syntax

We assume a finite set of data constructors, ranged over by L. The set of *expressions*, ranged over by e, is defined by:

$$\begin{array}{rcl} e \text{ (expressions)} & ::= & s \mid f(\widetilde{s}) \mid \texttt{fail} \mid \texttt{if } s \texttt{ then } e_1 \texttt{ else } e_2 \\ & & \mid \texttt{let } x = e_1 \texttt{ in } e_2 \\ & & \mid \texttt{match } s \texttt{ with } \{L_1(\widetilde{x}_1) \to e_1, \dots, L_k(\widetilde{x}_k) \to e_k\} \\ s \text{ (simple expressions)} & ::= & x \mid n \mid s_1 + s_2 \mid L(s_1, \dots, s_k) \\ D \text{ (programs)} & ::= & \{f_1(\widetilde{x}_1) = e_1, \dots, f_k(\widetilde{x}_k) = e_k\} \end{array}$$

The syntax of expressions above is fairly standard. A simple expression denotes an integer or a recursive data structure; we represent Booleans as integers, where non-zero integers are considered `true` and 0 is considered `false`. We write $\widetilde{\cdot}$ for a sequence; for example, $\widetilde{s}$ denotes a sequence of simple expressions $s_1, \dots, s_k$. For a technical convenience, the arguments of a function call $f(\widetilde{s})$ are restricted to simple expressions; this is not a fundamental restriction, $f\,e$ can be expressed by `let` $x = e$ `in` $f\,x$. The expression `fail` is a special command to indicate an error; the purpose of our refinement type system introduced later is to guarantee that `fail` does not occur during the execution of any well-typed program. As demonstrated in the examples below, the expression `fail` is often used to express the specification of a program. The conditional expression `if` s `then` e_1 `else` e_2 evaluates e_2 if the value of s is 0 and evaluates e_1 otherwise. The match expression `match` s `with` $\{L_1(\widetilde{x}_1) \to e_1, \dots, L_k(\widetilde{x}_k) \to e_k\}$ evaluates $[\widetilde{v}_i/\widetilde{x}_i]e_i$ if the value of s is $L_i(\widetilde{v}_i)$. For the sake of simplicity, we have only $+$ as an operator on integers, but other standard primitives ($-$, $\times$, $<$, $=$, ...) can be incorporated with no difficulty, and used in examples.

A program D is a set of (mutually recursive) function definitions. We assume that the set $\{f_1, \dots, f_k\}$ of function names contains `main`, the name of the "main" function.

2.2 Typing

We introduce a simple (monomorphic) type system, and require that programs and expressions are well-typed in the type system.

[1] The restriction to first-order programs is just for the sake of simplicity; our refinement type system can be easily extended for higher-order functions in a standard manner.

We assume a finite set $\mathcal{D}$ of (names of) recursive data types, ranged over by $\mathtt{d}$. The set of (simple) types, ranged over by κ, is defined by:

$$\begin{aligned}\kappa \text{ (simple types)} &::= \ b \mid (b_1, \dots, b_k) \to b \\ b \text{ (base types)} &::= \ \mathtt{int} \mid \mathtt{d}\end{aligned}$$

Here, a type of the form $(b_1, \dots, b_k) \to \mathtt{d}$ is called a constructor type. When $k = 1$, we just write $b \to \mathtt{d}$ for $(b) \to \mathtt{d}$. To distinguish simple types from refinement types introduced later, we sometimes call simple types *sorts*.

A *constructor environment*, written $\mathcal{C}$, is a map from the set of data constructor to the set of constructor types. A (simple) type environment, written $\mathcal{K}$, is a map from a finite set of variables to types. The type judgment relations $\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} e : \kappa$ and $\mathcal{C} \vdash_{\mathtt{ST}} D : \mathcal{K}$ are defined by the typing rules in Fig. 1.

Henceforth, we consider only expressions e and programs D such that $\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} e : \kappa$ and $\mathcal{C} \vdash_{\mathtt{ST}} D : \mathcal{K}$ for some $\mathcal{C}, \mathcal{K}$. As usual, programs well-typed in the simple type system do not get stuck; however, they may be reduced to the error state $\mathtt{fail}$.

In the rest of this paper, we further impose the following restriction on constructor types: for each constructor type $\mathcal{C}(L) = (b_1, \dots, b_k) \to \mathtt{d}$, we require that $\{b_1, \dots, b_k\} \subseteq \{\mathtt{int}, \mathtt{d}\}$. Thus, we forbid a constructor type like $(\mathtt{int}, \mathtt{d}_1) \to \mathtt{d}_2$ with $\mathtt{d}_1 \neq \mathtt{d}_2$. We permute argument types and normalize each constructor type to the form $(\mathtt{int}^k, \mathtt{d}^\ell) \to \mathtt{d}$. Again, the restriction is just for the sake of simplicity of the discussions in later sections. We write $\mathcal{C}_{\mathtt{d}}$ for the restriction of $\mathcal{C}$ on type $\mathtt{d}$, $\{L : \kappa \in \mathcal{C} \mid \kappa \text{ is of the form } (\widetilde{b}) \to \mathtt{d}\}$. Note that $\mathcal{C}$ can be decomposed to the disjoint union of maps $\mathcal{C}_{\mathtt{d}_1} \uplus \cdots \uplus \mathcal{C}_{\mathtt{d}_k}$. For the integer list type $\mathtt{ilist}$ discussed in Sect. 1, $\mathcal{C}_{\mathtt{ilist}} = \{\mathtt{Nil} \mapsto (\,) \to \mathtt{ilist}, \mathtt{Cons} \mapsto (\mathtt{int}, \mathtt{ilist}) \to \mathtt{ilist}\}$.

2.3 Operational Semantics

We define a small-step semantics of the language. The sets of evaluation contexts and values, respectively ranged over by E and v, are defined by:

$$\begin{aligned} E ::= &\ [\,] \mid E + s \mid n + E \mid L(\widetilde{v}, E, \widetilde{s}) \mid f(\widetilde{v}, E, \widetilde{s}) \mid \mathtt{if}\ E\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \\ &\mid \mathtt{let}\ x = E\ \mathtt{in}\ e \mid \mathtt{match}\ E\ \mathtt{with}\ \{L_1(\widetilde{x}_1) \to e_1, \dots, L_k(\widetilde{x}_k) \to e_k\} \\ v ::= &\ n \mid L(v_1, \dots, v_k) \end{aligned}$$

The reduction relation $e \longrightarrow_D e'$ on (closed) expressions is defined by the rules in Fig. 2. The expression $[\widetilde{v}/\widetilde{x}]e$ (which is an abbreviated form of $[v_1/x_1, \dots, v_k/x_k]e$) denotes the expression obtained from e by substituting $\widetilde{v}$ for $\widetilde{x}$. We write $\longrightarrow_D^*$ for the reflexive and transitive closure of $\longrightarrow_D$. We sometimes omit the subscript D and just write $\longrightarrow$ and $\longrightarrow^*$ for $\longrightarrow_D$ and $\longrightarrow_D^*$ respectively.

For a program D such that $\mathcal{C} \vdash_{\mathtt{ST}} D : \mathcal{K}$ and $\mathcal{K}(\mathtt{main}) = (b_1, \dots, b_k) \to \mathtt{int}$, we say D is *safe* if there exist no $v_1 : b_1, \dots, v_k : b_k$ and E such that $\mathtt{main}(v_1, \dots, v_k) \longrightarrow_D^* E[\mathtt{fail}]$. In the rest of this paper, we shall develop a

$$\frac{\mathcal{K}(x) = \kappa}{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} x : \kappa} \quad \text{(ST-Var)}$$

$$\frac{}{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} n : \mathtt{int}} \quad \text{(ST-Int)}$$

$$\frac{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} s_1 : \mathtt{int} \qquad \mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} s_2 : \mathtt{int}}{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} s_1 + s_2 : \mathtt{int}} \quad \text{(ST-Plus)}$$

$$\frac{\mathcal{C}(L) = (b_1, \dots, b_k) \to \mathtt{d} \qquad \mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} s_i : b_i \text{ for each } i \in \{1, \dots, k\}}{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} L(s_1, \dots, s_k) : \mathtt{d}} \quad \text{(ST-DC)}$$

$$\frac{\mathcal{K}(f) = (b_1, \dots, b_k) \to b \qquad \mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} s_i : b_i \text{ for each } i \in \{1, \dots, k\}}{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} f(s_1, \dots, s_k) : b} \quad \text{(ST-App)}$$

$$\frac{}{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} \mathtt{fail} : \mathtt{int}} \quad \text{(ST-Fail)}$$

$$\frac{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} s : \mathtt{int} \qquad \mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} e_1 : b \qquad \mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} e_2 : b}{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} \mathtt{if}\ s\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : b} \quad \text{(ST-If)}$$

$$\frac{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} e_1 : b_1 \qquad \mathcal{C}; \mathcal{K}, x : b_1 \vdash_{\mathtt{ST}} e_2 : b}{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : b} \quad \text{(ST-Let)}$$

$$\frac{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} s : \mathtt{d} \qquad \mathcal{C}(L_i) = (\widetilde{b_i}) \to \mathtt{d} \qquad \mathcal{C}; \mathcal{K}, \widetilde{x}_i : \widetilde{b}_i \vdash_{\mathtt{ST}} e_i : b \text{ for each } i \in \{1, \dots, k\}}{\mathcal{C}; \mathcal{K} \vdash_{\mathtt{ST}} \mathtt{match}\ s\ \mathtt{with}\ \{L_1(\widetilde{x}_1) \to e_1, \dots, L_k(\widetilde{x}_k) \to e_k\} : b} \quad \text{(ST-Match)}$$

$$\frac{\begin{array}{c}\mathcal{K} = (f_1 : (\widetilde{b}_1) \to b'_1, \dots, f_k : (\widetilde{b}_k) \to b'_k) \\ \mathcal{C}; \mathcal{K}, \widetilde{x}_i : \widetilde{b}_i \vdash_{\mathtt{ST}} e_i : b'_i \text{ for each } i \in \{1, \dots, k\}\end{array}}{\mathcal{C} \vdash_{\mathtt{ST}} \{f_1\ (\widetilde{x}_1) = e_1, \dots, f_k\ (\widetilde{x}_k) = e_k\} : \mathcal{K}} \quad \text{(ST-Prog)}$$

Fig. 1. Simple Type System

$$E[n_1 + n_2] \longrightarrow_D E[n] \text{ (if } n \text{ is the sum of } n_1 \text{ and } n_2) \quad \text{(E-Plus)}$$
$$E[f(\widetilde{v})] \longrightarrow_D E[[\widetilde{v}/\widetilde{x}]e] \text{ (if } f(\widetilde{x}) = e \in D) \quad \text{(E-Call)}$$
$$E[\mathtt{if}\ n\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2] \longrightarrow_D E[e_1] \text{ (if } n \neq 0) \quad \text{(E-IfT)}$$
$$E[\mathtt{if}\ 0\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2] \longrightarrow_D E[e_2] \quad \text{(E-IfF)}$$
$$E[\mathtt{match}\ L_i(\widetilde{v})\ \mathtt{with}\ \{L_1(\widetilde{x}_1) \to e_1, \dots, L_k(\widetilde{x}_k) \to e_k\}] \longrightarrow_D E[[\widetilde{v}/\widetilde{x}_i]e_i] \quad \text{(E-Match)}$$

Fig. 2. Reduction Rules

refinement type system that guarantees the safety of any well-typed program, and an automated procedure for proving the well-typedness, hence the safety of a given program. Note that the safety of a program does not imply the termination of the program; termination verification, for which various techniques [8,9] are available, is outside the scope of this paper.

Example 1. The program D_1 defined below declares function `range`, which takes an integer n and returns the list $[n, n-1, \dots, 1]$, and checks that the length of `range`(n) equals its argument n.

$$\begin{aligned}
D_1 = \{&\texttt{range}(n) = \texttt{if}\ n\ \texttt{then let}\ r = \texttt{range}(n-1)\ \texttt{in Cons}(n, r)\\
&\qquad\qquad\quad \texttt{else Nil}(),\\
&\texttt{len}(l) = \texttt{match}\ l\ \texttt{with}\ \{\texttt{Nil}() \to 0,\ \texttt{Cons}(n, l') \to 1 + \texttt{len}(l')\},\\
&\texttt{main}(n) = \texttt{let}\ r = \texttt{range}(n)\ \texttt{in let}\ l = \texttt{len}(r)\ \texttt{in}\\
&\qquad\qquad\quad \texttt{if}\ n \neq l\ \texttt{then fail else}\ 0\}
\end{aligned}$$

The evaluation of $\texttt{main}(n)$ terminates without failure if $n \geq 0$, and falls into an infinite loop if $n < 0$. □

Example 2. The following program D_2 focuses on function `isort`, which sorts a list in the ascending order by the insertion sort algorithm, and checks that its return value is sorted.

```
D₂ = {gen(n) = if n then Cons(∗, gen(n − 1)) else Nil(),
      insert(x, l) = match l with {
          Nil() → Cons(x, Nil()),
          Cons(y, l′) → if x < y then Cons(x, l) else Cons(y, insert(x, l′))
      },
      isort(l) = match l with {
          Nil() → Nil(), Cons(n, l′) → insert(n, isort(l′))
      },
      is_sorted_rec(x, l) = match l with {
          Nil() → 1,
          Cons(y, l′) → if x ≤ y then is_sorted_rec(y, l′) else 0
      },
      is_sorted(l) = match l with {
          Nil() → 1, Cons(n, l′) → is_sorted_rec(n, l′)
      },
      main(n) = let s = is_sorted(isort(gen(n))) in
                if s then 0 else fail
    }
```

The term $*$ indicates a non-deterministic integer value, omitted in the formal syntax for the sake of simplicity. The function `insert` constitutes a part of the insertion sort, which takes x and a sorted list l and returns a sorted list that consists of x and the elements of l. The function `is_sorted` returns 1 if the given list is sorted in the ascending order, and 0 otherwise. □

Example 3. The type `itree` for binary trees with integer values is defined with $\mathcal{C}_{\texttt{itree}} = \{\texttt{Leaf} \mapsto (\,) \to \texttt{itree}, \texttt{Node} \mapsto (\texttt{int}, \texttt{itree}, \texttt{itree}) \to \texttt{itree}\}$. The following program D_3 generates a random tree with a given size, and verifies that the generated tree has the given size as expected.

```
D_3 = { gen_tree(n) =
          if n then
            let m = * in let ℓ = gen_tree(m) in
            let r = gen_tree(n − 1 − m) in Node(*, l, r)
          else Nil(),
        size(t) = match t with {
            Leaf() → 0,
            Node(_, ℓ, r) → 1 + size(ℓ) + size(r)
        },
        main(n) = let s = size(gen_tree(n)) in
                  if s ≠ n then fail else 0
      }.
```

If $n \neq 0$,[2] gen_tree(n) picks a number m, and returns a tree of size n, consisting of the left child of size m and the right child of size $n - 1 - m$. Function size calculates the tree size (the number of nodes except leaves). □

3 A Parameterized Refinement Type System

This section introduces a refinement type system that guarantees the safety of well-typed programs.

3.1 Refinement Types

The syntax of *parameterized recursive refinement types*, ranged over by τ, is defined by:

$$\begin{aligned}
\tau \text{ (types)} &::= \{\beta \mid \varphi\} \mid \{(\beta_1, \dots, \beta_k) \mid \varphi'\} \to \{\beta \mid \varphi\} \\
\beta \text{ (type patterns)} &::= \delta[y_1, \dots, y_n] \\
\delta \text{ (raw types)} &::= \mathtt{int} \mid \mathtt{d}\langle n; (P_1, F_1), \dots, (P_k, F_k)\rangle \\
P \text{ (predicates)} &::= \lambda(\widetilde{y}).\varphi
\end{aligned}$$

Here, φ denotes a formula over integer arithmetic, and F denotes a function on integer tuples; we do not fix the precise syntax of φ and F, but assume that standard arithmetic and logical operators are available. In $\delta[y_1, \dots, y_n]$, (i) $n = 1$ if the *raw type* δ is int, and (ii) $n = m$ if $\delta = \mathtt{d}\langle m; (P_1, F_1), \dots, (P_k, F_k)\rangle$. Intuitively, $\{\mathtt{int}[x] \mid \varphi\}$ is the type of an integer x that satisfies φ. The type $\{(\beta_1, \dots, \beta_k) \mid \varphi'\} \to \{\beta \mid \varphi\}$ describes a function or a constructor that takes arguments of types $\beta_1, \dots, \beta_k$ that satisfy φ', and returns a value of type $\{\beta \mid \varphi\}$. For example, $\{(\mathtt{int}[x]) \mid x > 0\} \to \{\mathtt{int}[y] \mid y > x\}$ describes a function that takes a positive integer x as an argument and returns an integer greater than

[2] Actually, gen_tree(n) will not terminate if $n < 0$, but that does not concern us here since we are interested in only the safety property.

x. As this example indicates, the variables occurring in the part $(\beta_1, \ldots, \beta_k)$ are bound in $\{(\beta_1, \ldots, \beta_k) \mid \varphi'\} \to \{\beta \mid \varphi\}$, and may occur in φ' and φ. As usual, we allow implicit renaming of bound variables. We often write $\delta^![s_1, \ldots, s_n]$ for $\{\delta[y_1, \ldots, y_n] \mid y_1 = s_1 \wedge \cdots \wedge y_n = s_n\}$; we sometimes omit the superscript ! when there is no danger of confusion.

Refinement types for datatypes are more involved. For each (simple) datatype $\mathtt{d}$ with $\mathcal{C}_{\mathtt{d}} = \{L_1 : (\mathtt{int}^{\ell_1}, \mathtt{d}^{m_1}) \to \mathtt{d}, \ldots, L_k : (\mathtt{int}^{\ell_k}, \mathtt{d}^{m_k}) \to \mathtt{d}\}$, we consider refinement types of the form:

$$\{\mathtt{d}\langle n; (P_1, F_1), \ldots, (P_k, F_k)\rangle[y_1, \ldots, y_n] \mid \varphi\}.$$

Here, n denotes the number of integer parameters $y_1, \ldots, y_n$, and (P_i, F_i) is a pair of a predicate and a function corresponding to the constructor L_i. The above type denotes a data structure constructed from $L_1, \ldots, L_k$, by assigning the following type to L_i.

$$\{(\mathtt{int}[x_1], \ldots, \mathtt{int}[x_{\ell_i}], \delta[\widetilde{y}_1], \ldots, \delta[\widetilde{y}_{m_i}]) \mid P_i(\widetilde{x}, \widetilde{y}_1, \ldots, \widetilde{y}_{m_i})\} \\ \to \delta^![F_i(\widetilde{x}, \widetilde{y}_1, \ldots, \widetilde{y}_{m_i})]$$

Here, δ denotes $\mathtt{d}\langle n; (P_1, F_1), \ldots, (P_k, F_k)\rangle$, $\widetilde{x} = x_1, \ldots, x_{\ell_i}$, and $\widetilde{y}_i = y_{i,1}, \ldots, y_{i,n}$. Thus, the arity of the predicate P_i and the function F_i is $\ell_i + m_i n$, and F_i returns an n-tuple of integers. Recall that the part $\delta^![F_i(\widetilde{x}, \widetilde{y}_1, \ldots, \widetilde{y}_{m_i})]$ should be considered an abbreviated form of $\{\delta[z_1, \ldots, z_n] \mid (z_1, \ldots, z_n) = F_i(\widetilde{x}, \widetilde{y}_1, \ldots, \widetilde{y}_{m_i})\}$. Note that P_i and F_i take only integers as their arguments; thus information about recursive data structures is abstracted to integers by the type system.

For example, ilistL in Sect. 1 is expressed as

$$\mathtt{ilist}\langle 1; (\lambda().\mathtt{true}, \lambda().0), (\lambda(x, y).\mathtt{true}, \lambda(x, y).y + 1)\rangle,$$

and the constructors Nil and Cons are given the following types:

$$\mathtt{Nil} : () \to \mathtt{ilistL}^![0]$$
$$\mathtt{Cons} : (\mathtt{int}[x], \mathtt{ilistL}[y]) \to \mathtt{ilistL}^![y + 1].$$

Note that the argument type of Cons is

$$\{(\mathtt{int}[x], \mathtt{ilistL}[y]) \mid (\lambda(x, y).\mathtt{true})(x, y)\} \equiv \{(\mathtt{int}[x], \mathtt{ilistL}[y]) \mid \mathtt{true}\},$$

which has been abbreviated to $(\mathtt{int}[x], \mathtt{ilistL}[y])$.

As another example, recall ilistS in Sect. 1. It is expressed as:

$$\mathtt{ilist}\langle 2; (\lambda(\,).\mathtt{true}, \lambda(\,).(0, 0)), \\ (\lambda(x, y_1, y_2).(y_1 > 0 \Rightarrow x \le y_2), \lambda(x, y_1, y_2).(1, x))\rangle,$$

and the constructors are given the following types:

$$\mathtt{Nil} : (\,) \to \mathtt{ilistS}^![0, 0]$$
$$\mathtt{Cons} : \{(\mathtt{int}[x], \mathtt{ilistS}[y_1, y_2]) \mid y_1 > 0 \Rightarrow x \le y_2\} \to \mathtt{ilistS}^![1, x].$$

Remark 1. If we are interested in proving that a sorting function takes an integer list as an argument and returns a sorted list that is a *permutation* of the argument, we need to parameterize the list type also with information about the elements of a list. One way to do so would be to introduce the type $\texttt{ilistP}[y_1, y_2, y_3]$ of a list of length y_1 that contains y_3 occurrences of the element y_2, and the type $\texttt{ilistSP}[y_1, y_2, y_3, y_4]$ of a sorted list (of type $\texttt{ilistS}[y_1, y_2]$) containing y_4 occurrences of the element y_3. Then the type of a sorting function can be expressed as: $\{\texttt{ilistP}[y_1, y_2, y_3] \mid \texttt{true}\} \rightarrow \{\texttt{ilistSP}[y_1, z, y_2, y_3] \mid \texttt{true}\}$. $\square$

3.2 Typing

We define the type judgment relations $\mathcal{C}; \Gamma; \varphi \vdash e : \tau$ and $\mathcal{C} \vdash D : \Gamma$ for expressions and programs by the typing rules in Fig. 3. Here, $\mathcal{C}$ is a constructor type environment as before, and Γ maps each variable (including a function name) to its type. The type bindings on integer types and datatypes are restricted to the form $x : \{\beta \mid \texttt{true}\}$, so we just write $x : \beta$. The conditions on variables of integer types and datatypes are instead accumulated in the part φ of the type environment. Type bindings on integer types are further restricted to $x : \texttt{int}[x]$; hence we sometimes just write $x : \texttt{int}$. In a type judgment $\mathcal{C}; \Gamma; \varphi \vdash e : \tau$, we implicitly require that all the types are well-formed; for example, φ and τ may contain only integer variables occurring in Γ (including those in the part β) as free variables. The definition of well-formedness is given in the longer version of this paper [13].

The type judgment $\mathcal{C}; \Gamma; \varphi \vdash e : \tau$ intuitively means that if each free variable in e has type $\Gamma(x)$ and satisfies the condition described by φ, then e is safely executed (without reaching $\texttt{fail}$), and either e diverges or evaluates to a value of type τ. In Fig. 3, $\models \varphi$ means that the formula φ is a valid formula of integer arithmetic.

We explain some key rules. The typing rules for expressions are fairly standard, except T-DC and T-Sub for datatypes. In T-App, we require that the β-part of the argument types matches between the function and actual arguments. The condition $\models \varphi \wedge (\bigwedge_{i=1}^{k} \varphi_i) \Rightarrow \varphi'$ requires that the condition φ' required by the function is met by the actual arguments. In rule T-Fail, the condition $\models \neg\varphi$ ensures that there exists no environment that makes φ hold, so that $\texttt{fail}$ is unreachable. In T-If, the branching condition is accumulated in the conditions for the then- and else-branches. In T-Let, the condition φ_1 on the value of e_1 is accumulated in the condition for e_2.

In rule T-DC, the third and fourth conditions require that the arguments of the constructor L_i has an appropriate type, and the fifth condition requires that they also satisfy the precondition P_i. The last premise ensures the post condition (represented by the function F_i) of the data constructor implies the condition φ' on the constructed data. Note that the "δ-part" may be locally chosen in the rule (thus, the constructor L_i is polymorphic on $\langle n; (P_1, F_1), \ldots, (P_k, F_k)\rangle$, and that part may be instantiated for each occurrence of the constructor), but that the same δ must be used among $L_i(s_1, \ldots, s_{\ell_i+m_i})$ and the components $s_{\ell_i+1}, \ldots, s_{\ell_i+m_i}$.

$$\frac{\Gamma(x)=\beta \qquad \models \varphi \Rightarrow \varphi'}{\mathcal{C};\Gamma;\varphi \vdash x : \{\beta \mid \varphi'\}} \quad \text{(T-Var)}$$

$$\frac{\mathcal{C};\mathtt{ST}(\Gamma) \vdash_{\mathtt{ST}} s : \mathtt{int}}{\mathcal{C};\Gamma;\varphi \vdash s : \{\mathtt{int}[x] \mid x = s\}} \quad \text{(T-Int)}$$

$$\frac{\begin{array}{c}\Gamma(f)=\{(\beta_1,\dots,\beta_k)\mid\varphi'\}\to\{\beta\mid\varphi_{\mathrm{r}}\}\\ \mathcal{C};\Gamma;\varphi\vdash s_i:\{\beta_i\mid\varphi_i\}\text{ for each } i\in\{1,\dots,k\}\\ \models\varphi\wedge(\bigwedge_{i=1}^{k}\varphi_i)\Rightarrow\varphi'\\ \models\varphi\wedge(\bigwedge_{i=1}^{k}\varphi_i)\wedge\varphi_{\mathrm{r}}\Rightarrow\varphi'_{\mathrm{r}}\end{array}}{\mathcal{C};\Gamma;\varphi\vdash f(s_1,\dots,s_k):\{\beta\mid\varphi'_{\mathrm{r}}\}} \quad \text{(T-App)}$$

$$\frac{\models \neg\varphi}{\mathcal{C};\Gamma;\varphi \vdash \mathtt{fail} : \mathtt{int}} \quad \text{(T-Fail)}$$

$$\frac{\mathcal{C};\Gamma;\varphi\vdash s:\mathtt{int} \qquad \mathcal{C};\Gamma;\varphi\wedge s\neq 0\vdash e_1:\tau \qquad \mathcal{C};\Gamma;\varphi\wedge s=0\vdash e_2:\tau}{\mathcal{C};\Gamma;\varphi\vdash \mathtt{if}\ s\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2:\tau} \quad \text{(T-If)}$$

$$\frac{\mathcal{C};\Gamma;\varphi\vdash e_1:\{\beta\mid\varphi_1\} \qquad \mathcal{C};\Gamma,x:\beta;\varphi\wedge\varphi_1\vdash e_2:\tau}{\mathcal{C};\Gamma;\varphi\vdash \mathtt{let}\ x=e_1\ \mathtt{in}\ e_2:\tau} \quad \text{(T-Let)}$$

$$\frac{\begin{array}{c}\mathcal{C}_{\mathtt{d}}=\{L_1:(\mathtt{int}^{\ell_1},\mathtt{d}^{m_1})\to\mathtt{d},\dots,L_k:(\mathtt{int}^{\ell_k},\mathtt{d}^{m_k})\to\mathtt{d}\}\\ \delta=\mathtt{d}\langle n;(P_1,F_1),\dots,(P_k,F_k)\rangle\\ \mathcal{C};\Gamma;\varphi\vdash s_j:\{\mathtt{int}[x_j]\mid\varphi_j\}\text{ for each } j\in\{1,\dots,\ell_i\}\\ \mathcal{C};\Gamma;\varphi\vdash s_{\ell_i+j}:\{\delta[\widetilde{y}_j]\mid\varphi_{\ell_i+j}\}\text{ for each } j\in\{1,\dots,m_i\}\\ \models\varphi\wedge(\bigwedge_{j=1}^{\ell_i+m_i}\varphi_j)\Rightarrow P_i(x_1,\dots,x_{\ell_i},\widetilde{y}_1,\dots,\widetilde{y}_{m_i})\\ \models\varphi\wedge(\bigwedge_{j=1}^{\ell_i+m_i}\varphi_j)\wedge(\widetilde{y})=F_i(x_1,\dots,x_{\ell_i},\widetilde{y}_1,\dots,\widetilde{y}_{m_i})\Rightarrow\varphi'\end{array}}{\mathcal{C};\Gamma;\varphi\vdash L_i(s_1,\dots,s_{\ell_i+m_i}):\{\delta[\widetilde{y}]\mid\varphi'\}} \quad \text{(T-DC)}$$

$$\frac{\begin{array}{c}\mathcal{C}_{\mathtt{d}}=\{L_1:(\mathtt{int}^{\ell_1},\mathtt{d}^{m_1})\to\mathtt{d},\dots,L_k:(\mathtt{int}^{\ell_k},\mathtt{d}^{m_k})\to\mathtt{d}\}\\ \delta=\mathtt{d}\langle n;(P_1,F_1),\dots,(P_k,F_k)\rangle\\ \mathcal{C};\Gamma;\varphi\vdash s:\{\delta[\widetilde{y}]\mid\varphi_0\}\\ \Gamma'_i=\Gamma,x_1:\mathtt{int}[x_1],\dots,x_{\ell_i}:\mathtt{int}[x_{\ell_i}],x_{\ell_i+1}:\delta[\widetilde{y}_1],\dots,x_{\ell_i+m_i}:\delta[\widetilde{y}_{m_i}]\\ \varphi'_i=\varphi\wedge P_i(x_1,\dots,x_{\ell_i},\widetilde{y}_1,\dots,\widetilde{y}_{m_i})\wedge[F_i(x_1,\dots,x_{\ell_i},\widetilde{y}_1,\dots,\widetilde{y}_{m_i})/\widetilde{y}]\varphi_0\\ \mathcal{C};\Gamma'_i;\varphi'_i\vdash e_i:\tau\text{ for each } i\in\{1,\dots,k\}\end{array}}{\mathcal{C};\Gamma;\varphi\vdash \mathtt{match}\ s\ \mathtt{with}\ \{L_1(\widetilde{x}_1)\to e_1,\dots,L_k(\widetilde{x}_k)\to e_k\}:\tau} \quad \text{(T-Match)}$$

$$\frac{\mathcal{C};\Gamma;\varphi\vdash e:\{\beta\mid\varphi_1\} \qquad \models\varphi\wedge\varphi_1\Rightarrow\varphi_2}{\mathcal{C};\Gamma;\varphi\vdash e:\{\beta\mid\varphi_2\}} \quad \text{(T-Sub)}$$

$$\frac{\begin{array}{c}\Gamma=(f_1:\{(\widetilde{\beta}_1)\mid\varphi_1\}\to\{\beta'_1\mid\varphi'_1\},\dots,f_k:\{(\widetilde{\beta}_k)\mid\varphi_k\}\to\{\beta'_k\mid\varphi'_k\})\\ \mathcal{C};\Gamma,\widetilde{x}_i:\widetilde{\beta}_i;\varphi_i\vdash e_i:\{\beta'_i\mid\varphi'_i\}\text{ for each } i\in\{1,\dots,k\}\end{array}}{\mathcal{C}\vdash\{f_1(\widetilde{x}_1)=e_1,\dots,f_k(\widetilde{x}_k)=e_k\}:\Gamma} \quad \text{(T-Prog)}$$

Fig. 3. Refinement Type System

In rule T-Match, the type environment Γ'_i for the subexpression e_i is obtained from Γ by adding type bindings for the variables $\widetilde{x}_i$ (see the fourth line of the premises). The condition φ'_i (defined on the fifth line) is obtained by strengthening the condition φ with information that s matches $L_i(\widetilde{x}_i)$. Note that as in rule T-DC, the "δ-part" is shared among s and decomposed elements (bound to) $x_{\ell_i+1}, \ldots, x_{\ell_i+m_i}$. The rule T-Sub is for subsumption. We allow only the refinement condition to be weakened; for datatypes, the β-part (of the form $\mathtt{d}\langle n; (P_1, F_1), \ldots, (P_k, F_k)\rangle[\widetilde{y}]$) is fixed.

Example 4. Let us recall the program D_1 defined in Example 1. It is typed as $\mathcal{C} \vdash D_1 : \Gamma_0$, where Γ_0 consists of:

$$\begin{aligned}&\mathtt{range}\colon \mathtt{int}[n] \to \mathtt{ilistL}^{!}[n],\\ &\mathtt{len}\colon \mathtt{ilistL}[n] \to \mathtt{int}^{!}[n],\\ &\mathtt{main}\colon \mathtt{int}[n] \to \{\mathtt{int}[x] \mid \mathtt{true}\}.\end{aligned}$$

Below we focus on the definition of the function range, and show how to derive $\mathcal{C}; \Gamma_1; \mathtt{true} \vdash \mathtt{if}\ n\ \mathtt{then}\ e_2\ \mathtt{else}\ \mathtt{Nil}() : \mathtt{ilistL}^{!}[n]$ (which is required for deriving $\mathcal{C} \vdash D_1 : \Gamma_0$), where

$$\begin{aligned}\Gamma_1 &= (\Gamma_0, n : \mathtt{int})\\ e_2 &= (\mathtt{let}\ r = \mathtt{range}(n-1)\ \mathtt{in}\ \mathtt{Cons}(n, r))\\ \mathtt{ilistL} &= \mathtt{ilist}\langle 1; (\lambda().\mathtt{true}, \lambda().0), (\lambda(x,y).\mathtt{true}, \lambda(x,y).y+1)\rangle.\end{aligned}$$

First, the type of $\mathtt{range}(n-1)$ in the body is derived as follows.

$$\dfrac{\begin{array}{c}\Gamma_1(\mathtt{range}) = \{\mathtt{int}[m] \mid \mathtt{true}\} \to \{\mathtt{ilistL}[y] \mid y = m\}\\ \mathcal{C}; \Gamma_1; n \neq 0 \vdash n-1 : \{\mathtt{int}[m] \mid m = n-1\}\\ \models m = n-1 \Rightarrow \mathtt{true}\\ \models m = n-1 \wedge y = m \Rightarrow y = n-1\end{array}}{\mathcal{C}; \Gamma_1; n \neq 0 \vdash \mathtt{range}(n-1) : \{\mathtt{ilistL}[y] \mid y = n-1\}.}\ (\text{T-App})$$

Second, the expression $\mathtt{Cons}(n, r)$ is typed as:

$$\dfrac{\begin{array}{c}\mathcal{C}; \Gamma_2; \varphi_2 \vdash n : \{\mathtt{int}[x_1] \mid \mathtt{true}\}\\ \mathcal{C}; \Gamma_2; \varphi_2 \vdash r : \{\mathtt{ilistL}[y_1] \mid y_1 = n-1\}\\ \models \varphi_2 \Rightarrow (\lambda(x,y).\mathtt{true})(x_1, y_1)\\ \models \varphi_2 \wedge y_1 = n-1 \wedge z = (\lambda(x,y).y+1)(x_1,y_1) \Rightarrow z = n\end{array}}{\mathcal{C}; \Gamma_2; \varphi_2 \vdash \mathtt{Cons}(n, r) : \{\mathtt{ilistL}[z] \mid z = n\},}\ (\text{T-DC})$$

where $\Gamma_2 = (\Gamma_1, r : \mathtt{ilistL}[q])$ and $\varphi_2 = (n \neq 0 \wedge q = n-1)$. Finally, using the judgments above, we obtain:

$$\dfrac{\dfrac{\begin{array}{c}\mathcal{C}; \Gamma_1; n \neq 0 \vdash \mathtt{range}(n-1) : \mathtt{ilistL}^{!}[n-1]\\ \mathcal{C}; \Gamma_2; \varphi_2 \vdash \mathtt{Cons}(n, r) : \mathtt{ilistL}^{!}[n]\end{array}}{\mathcal{C}; \Gamma_1; n \neq 0 \vdash e_2 : \mathtt{ilistL}^{!}[n]} \quad \dfrac{\begin{array}{c}\models n = 0 \Rightarrow (\lambda().\mathtt{true})()\\ \models n = 0 \wedge y = (\lambda().0)() \Rightarrow y = n\end{array}}{\mathcal{C}; \Gamma_1; n = 0 \vdash \mathtt{Nil}() : \mathtt{ilistL}^{!}[n]}}{\mathcal{C}; \Gamma_1; \mathtt{true} \vdash \mathtt{if}\ n\ \mathtt{then}\ e_2\ \mathtt{else}\ \mathtt{Nil}() : \mathtt{ilistL}^{!}[n].}$$

□

Our type system can also deal with properties on trees, as demonstrated in the following example.

Example 5. Recall the program D_3 given in Example 3. It is typed as $\mathcal{C} \vdash D_3 : \mathit{\Gamma}_0$, where $\mathit{\Gamma}_0$ is:

$$\begin{aligned}\mathit{\Gamma}_0 = \{&\texttt{gen\_tree}\colon \texttt{int}[n] \to \texttt{itreeZ}^{!}[n],\\ &\texttt{size}\colon \texttt{itreeZ}[n] \to \texttt{int}^{!}[n],\\ &\texttt{main}\colon \texttt{int}[n] \to \{\texttt{int}[x] \mid \texttt{true}\}\}.\end{aligned}$$

Here, $\texttt{itreeZ} = \texttt{itree}\langle 1; (\lambda().\texttt{true}, \lambda().0), (\lambda(x, y_1, y_2).\texttt{true}, \lambda(x, y_1, y_2).y_1 + y_2 + 1)\rangle$. Intuitively, $\texttt{itreeZ}[n]$ is the type of trees with n nodes. The expression $\texttt{Node}(*, \ell, r)$ in the definition of the function $\texttt{gen\_tree}$ is typed by:

$$\dfrac{\begin{array}{c}\mathcal{C}; \mathit{\Gamma}_1; \varphi_1 \vdash * : \{\texttt{int}[x_1] \mid \texttt{true}\}\\ \mathcal{C}; \mathit{\Gamma}_1; \varphi_1 \vdash \ell : \{\texttt{itreeZ}[y_1] \mid y_1 = m\}\\ \mathcal{C}; \mathit{\Gamma}_1; \varphi_1 \vdash r : \{\texttt{itreeZ}[y_2] \mid y_2 = n - 1 - m\}\\ \models \varphi_1 \Rightarrow (\lambda(x, y_1, y_2).\texttt{true})(x_1, y_1, y_2)\\ \models \varphi_1 \wedge y_1 = m \wedge y_2 = n - 1 - m \wedge z = y_1 + y_2 + 1 \Rightarrow z = n\end{array}}{\mathcal{C}; \mathit{\Gamma}_1; \varphi_1 \vdash \texttt{Node}(*, \ell, r) : \{\texttt{itreeZ}[z] \mid z = n\},}\ (\text{T-DC})$$

where

$$\begin{aligned}\mathit{\Gamma}_1 &= (\mathit{\Gamma}_0, n : \texttt{int}, m : \texttt{int}, \ell : \texttt{itreeZ}[m], r : \texttt{itreeZ}[n - 1 - m])\\ \varphi_1 &= (n \neq 0).\end{aligned}$$

The last premise ($\models \varphi_1 \wedge y_1 = m \wedge y_2 = n - 1 - m \wedge z = y_1 + y_2 + 1 \Rightarrow z = n$) uses the function $\lambda(x, y_1, y_2).y_1 + y_2 + 1$ in $\texttt{itreeZ}$ to obtain an accumulated value for the tree size.

The $\texttt{match}$ expression in function $\texttt{size}$ is typed by:

$$\dfrac{\dfrac{\mathcal{C}; \mathit{\Gamma}_2; \varphi_1' \vdash 0 : \texttt{int}^{!}[0]}{\mathcal{C}; \mathit{\Gamma}_2; \varphi_1' \vdash 0 : \texttt{int}^{!}[n]}\ (\text{T-Sub}) \quad \dfrac{\dfrac{\vdots}{\mathcal{C}; \mathit{\Gamma}_2'; \varphi_2' \vdash e_3 : \texttt{int}^{!}[1 + y_1 + y_2]}}{\mathcal{C}; \mathit{\Gamma}_2'; \varphi_2' \vdash e_3 : \texttt{int}^{!}[n]}\ (\text{T-Sub})}{\mathcal{C}; \mathit{\Gamma}_2; \texttt{true} \vdash e_2 : \texttt{int}^{!}[n],}\ (\text{T-Match})$$

where

$$\begin{aligned}\mathit{\Gamma}_2 &= (\mathit{\Gamma}_0, t : \texttt{itreeZ}[n])\\ \mathit{\Gamma}_2' &= (\mathit{\Gamma}_2, \_ : \texttt{int}, \ell : \texttt{itreeZ}[y_1], r : \texttt{itreeZ}[y_2])\\ e_2 &= (\texttt{match}\ t\ \texttt{with}\ \{\texttt{Leaf}() \to 0, \texttt{Node}(\_, \ell, r) \to e_3\})\\ e_3 &= 1 + \texttt{size}(\ell) + \texttt{size}(r)\\ \varphi_1' &= (n = 0)\\ \varphi_2' &= (n = 1 + y_1 + y_2).\end{aligned}$$

□

Remark 2. It is sometimes too restrictive to fix the β-part in rule T-Sub. For example, the function `isort` of the program D_2 (defined in Example 2) is equivalent to the function $\texttt{isort}'$ defined below, which is obtained by substituting $\texttt{Nil}()$ in the match body of D_2 with l.

$$\begin{aligned}\texttt{isort}'(l) = \texttt{match}\ l\ \texttt{with}\ \{&\\ &\texttt{Nil}() \to l,\ \texttt{Cons}(n, l') \to \texttt{insert}(n, \texttt{isort}'(l'))\\ \}.&\end{aligned}$$

However, since l is returned directly, the argument and return types of $\texttt{isort}'$ share the same β-part. Therefore, our type system cannot express that $\texttt{isort}'$ converts an unsorted list to a sorted one. To relax the restriction, we need a more sophisticated version of the subtyping rule T-Sub, which would cause too much burden for the type inference procedure discussed in the next section. It is left for future work to overcome the problem above without incurring too much overhead for type inference. □

The following proposition states the soundness of the type system (recall the definition of safety in Sect. 2.3).

Proposition 1 (soundness). *Suppose* $\mathcal{C} \vdash D : \Gamma$, *with* $\Gamma(\texttt{main}) = \{(\beta_1, \ldots, \beta_k) \mid \texttt{true}\} \to \{\beta \mid \texttt{true}\}$. *Then, the program* D *is safe.*

The proposition follows from the soundness of a standard refinement type system without parameterization $\langle n; (P_1, F_1), \ldots, (P_k, F_k)\rangle$, as follows. Because only constructors are polymorphic on the part $\langle n; (P_1, F_1), \ldots, (P_k, F_k)\rangle$, if a program D is well-typed, then by annotating each occurrence of constructor L_i with the parameter $\langle n; (P_1, F_1), \ldots, (P_k, F_k)\rangle$, and treating the annotated constructor $L_i^{(\langle n;(P_1,F_1),\ldots,(P_k,F_k)\rangle)}$ as a new constructor, and the δ-part $\mathtt{d}\langle n; (P_1, F_1), \ldots, (P_k, F_k)\rangle$ as the name of a new datatype, we can obtain a program D' that is well-typed without the parameterization. The safety of D' follows from the soundness of a standard refinement type system (without parameterization); hence D is also safe.

Note that the completeness does not hold: there exists a program that is safe but not typable in our refinement type system. Beside the issue discussed in Remark 2, the sources of incompleteness include the restriction of the parameters of data types to integers. For example, consider the property of the append function: "a function takes two lists and returns the list obtained by appending two lists." In theory, it is possible to encode all the information of a list by using Gödel encoding, but that is not possible in practice, where we have to restrict the underlying integer arithmetic, e.g., to linear integer arithmetic.

4 Inferring Parameterized Refinement Types

This section describes a type inference procedure, which takes a program (without type annotations) and a constructor type environment as input, and checks whether the program is well-typed. The overall flow of the type inference procedure is shown in Fig. 4.

In Step 1, we first determine the raw type of each expression, with the values of the part $[n; \widetilde{(P, F)}]$ kept unknown. For example, given the program D_2 in Example 2, we infer:

$$\texttt{gen} : \texttt{int} \to \texttt{ilist}[\rho_1], \texttt{isort} : \texttt{ilist}[\rho_1] \to \texttt{ilist}[\rho_2],$$

where ρ_1 and ρ_2 are variables representing the part $[n; \widetilde{(P, F)}]$. (Note that the same variable ρ_1 is assigned to the return type of `gen` and the argument type of `isort`, since the return value of `gen` is passed to `isort`.) This is performed by using an ordinary unification-based type inference algorithm.

In Step 2, the part n and $\widetilde{F}$ of each raw type variable ρ_i is chosen, while the predicates $\widetilde{P}$ are kept unknown. In Step 3, we prepare predicate variables for the unknown predicates in raw types and refinement predicates, and reduce the typability problem to the satisfiability problem for constrained Horn clauses (CHCs) [1]. We then invoke an off-the-shelf CHC solver [2,4,7] to check whether the obtained CHCs are satisfiable. If so, we can conclude that the program is well-typed (and outputs inferred types); otherwise, we go back to Step 2 and refine the F-part of raw types, with an increased value of n.

In the rest of this section, we explain more details of Steps 2 and 3.

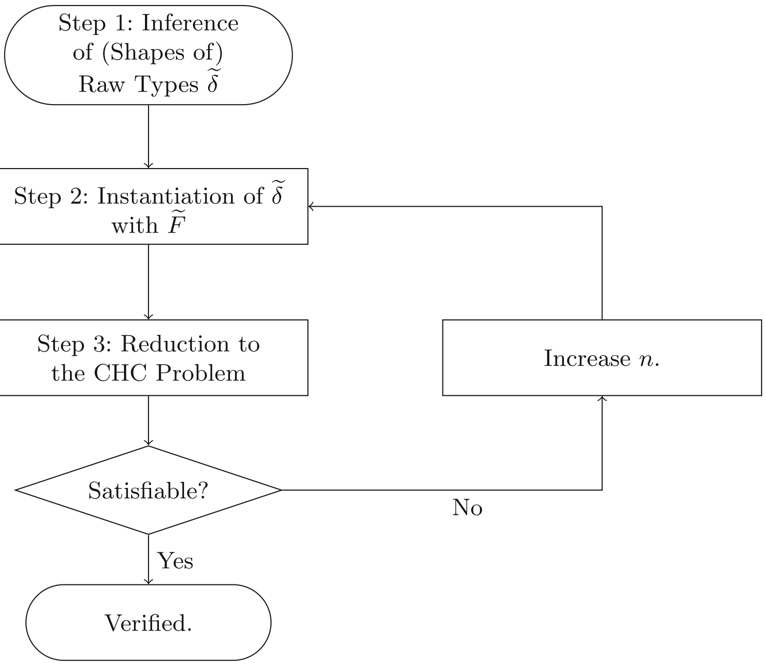
Step 1: Inference of (Shapes of) Raw Types $\widetilde{\delta}$
Step 2: Instantiation of $\widetilde{\delta}$ with $\widetilde{F}$
Step 3: Reduction to the CHC Problem
Increase n.
Satisfiable?
No
Yes
Verified.

Fig. 4. The flow of type inference

4.1 Step 2: Instantiation of Raw Types with $\widetilde{F}$

In Step 2, we determine the components n and $\widetilde{F}$ of δ.

For the sake of simplicity, the number of integer parameters n is shared by all types, and the functions $\widetilde{F}$ do not depend on δ but on $\mathtt{d}$. On the other hand, the predicate variables $\widetilde{P}$ are specific to δ. Thus, we explicitly write $\delta = \mathtt{d}\langle n; (P_{\delta,1}, F_{\mathtt{d},1}), \dots, (P_{\delta,k}, F_{\mathtt{d},k})\rangle$ here.

We choose n and $F_{\mathtt{d},j}$ as follows, to ensure that the precision of type inference is monotonically improved at each iteration. Suppose $\mathcal{C}_{\mathtt{d}} = \{L_1 : (\mathtt{int}^{\ell_1}, \mathtt{d}^{m_1}) \to \mathtt{d}, \dots, L_k : (\mathtt{int}^{\ell_k}, \mathtt{d}^{m_k}) \to \mathtt{d}\}$. Let us write $n^{(i)}$ and $F^{(i)}_{\mathtt{d},j}$ for the values of n and $F_{\mathtt{d},j}$ at the i-th iteration of the refinement loop in Fig. 4. At the $(i+1)$-th iteration, we pick $n' > 0$ and a tuple of functions $(F'_1, \dots, F'_k)$ with $F'_j \in \mathtt{int}^{\ell_j + m_j n'} \to \mathtt{int}^{n'}$ (that has not been chosen before) and set $n^{(i+1)}$ and $F^{(i+1)}_{\mathtt{d},j}$ as follows.

$$n^{(i+1)} := n^{(i)} + n'$$
$$F^{(i+1)}_{\mathtt{d},j} := \lambda(\widetilde{x}, \widetilde{y}_1, \dots, \widetilde{y}_{m_j}).(F^{(i)}(\widetilde{x}, \widetilde{y}'_1, \dots, \widetilde{y}'_{m_j}), F'_j(\widetilde{x}, \widetilde{y}''_1, \dots, \widetilde{y}''_{m_j})).$$

Here, $\widetilde{x}$ and $\widetilde{y}_j$ are sequences of variables of length ℓ_k and $n^{(i+1)}$ respectively, and $\widetilde{y}_j = \widetilde{y}'_j, \widetilde{y}''_j$ with $|\widetilde{y'}_j| = n^{(i)}$ and $|\widetilde{y}''_i| = n'$. For example, if $n^{(i)} = 1$ and $F^{(i)}_j(x, y_1, y_2) = x + y_1 + y_2$ with $n' = 1$ and $F'_j(x, y_1, y_2) = 1 + \max(y_1, y_2)$, then $F^{(i+1)}_j(x, y_{11}, y_{12}, y_{21}, y_{22}) = (x + y_{11} + y_{21}, 1 + \max(y_{12}, y_{22}))$.

Since the choice of $n^{(i)}$ and $F^{(i)}_{\mathtt{d},j}$ above ensures that the information carried by types monotonically increases, we can guarantee that our type inference procedure is *relatively complete* with respect to the (hypothetical[3]) completeness of the CHC solver used in Step 3, in the following sense. Let us assume that the language for describing functions of type $\bigcup_{j=1}^{\omega} \mathtt{int}^{l_i + m_i j} \to \mathtt{int}^j$ is recursively enumerable; for example, we can restrict functions to those expressible in linear integer arithmetic. Then we can enumerate all the tuples of functions and use the i-th tuple as $(F'_1, \dots, F'_k)$ above. Suppose that a program D is typable by using, as $F_{\mathtt{d},j}$, functions belonging to the language assumed above. Then, assuming that the CHC solver used in Step 3 below is complete, our procedure eventually terminates and outputs "Verified". (In other words, our procedure eventually terminates output "Verified", or gets stuck in Step 3 due to the incompleteness of the CHC solver.) This is because the functions required for typing D is eventually chosen and added to $F^{(i)}_{\mathtt{d},j}$.

For the sake of efficiency, the actual implementation imposes a further restriction on the function F'_j added at each iteration, at the sacrifice of relative completeness; see Sect. 5.1.

Remark 3. While we currently employ the same n for all data types, it can be effective to selectively add a parameter to an individual raw type, based on the unsatisfiable core returned from the solver in Step 3.

[3] Since the CHC satisfiability problem is undecidable in general, there is no complete CHC solver.

4.2 Step 3: Reduction to CHC Solving

In this step, we prepare predicate variables for the P-part of raw types and unknown refinement predicates φ, and construct a template of a type derivation tree. We then extract constraints on the predicate variables based on the typing rules. The extracted constraints consists of *constrained Horn clauses* (CHCs), of the following form:

$$\forall \widetilde{x}.(H \Leftarrow B_1 \wedge \cdots \wedge B_k),$$

where B_i and H are atomic constraints of the form $p(s_1, \ldots, s_\ell)$ or integer constraints ($s_1 \leq s_2$, $s_1 = s_2$,...). The program is well-typed (with the choice of n and $\widetilde{F}$ in the previous step), just if the CHCs are satisfiable, i.e., if there exists an assignment of predicates to predicate variables that make all the clauses valid. The latter problem (of CHC satisfiability) is undecidable in general, but there are various efficient solvers that work well for many inputs [2,4,7].

Since the reduction from refinement type inference to the CHC satisfiability problem is fairly standard (see, e.g., [2,18]), we sketch the reduction only informally, through an example.

Example 6. Let us recall the program D_1 in Example 1, and focus on the function `range`. When $n = 1$, we need to derive $\mathcal{C}; \Gamma_1; p_1(h) \vdash \texttt{if}\ h\ \texttt{then}\ e_2\ \texttt{else}\ \texttt{Nil}() : \tau_0$ (which is required in T-Prog for proving $\mathcal{C} \vdash D_1 : \Gamma_0$), where

- $\Gamma_0 = (\texttt{range}\colon \{\texttt{int}[h] \mid p_1(h)\} \to \{\texttt{ilistL}[i] \mid p_2(h,i)\}, \ldots)$,
- $\Gamma_1 = (\Gamma_0, h : \texttt{int})$,
- $e_2 = (\texttt{let}\ r = \texttt{range}(h-1)\ \texttt{in}\ \texttt{Cons}(h, r))$,
- $\tau_0 = \{\texttt{ilist}_1[i] \mid p_2(h,i)\}$, and
- $\texttt{ilist}_1 := \texttt{ilist}\langle 1; (p_3, \lambda().0), (p_4, \lambda(x,y).y+1)\rangle$.

The derivation for the judgment is of the form:

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ \mathcal{C}; \Gamma_1; p_1(h) \wedge h \neq 0 \vdash \texttt{range}(h-1) : \tau_3 \\ \mathcal{C}; \Gamma_1, r : \tau_3; \varphi_2 \vdash \texttt{Cons}(h,r) : \tau_0\end{array}}{\mathcal{C}; \Gamma_1; p_1(h) \wedge h \neq 0 \vdash e_2 : \tau_0} \quad \dfrac{\begin{array}{c}\models p_1(h) \wedge h = 0 \Rightarrow p_3() \\ \models p_1(h) \wedge h = 0 \wedge i = 0 \Rightarrow p_2(h,i)\end{array}}{\mathcal{C}; \Gamma_1; p_1(h) \wedge h = 0 \vdash \texttt{Nil}() : \tau_0}}{\mathcal{C}; \Gamma_1; p_1(h) \vdash \texttt{if}\ h\ \texttt{then}\ e_2\ \texttt{else}\ \texttt{Nil}() : \tau_0.}$$

where $\varphi_2 = (p_1(h) \wedge h \neq 0 \wedge p_5(h,j))$ and $\tau_3 = \{\texttt{ilistL}[j] \mid p_5(h,j)\}$. From the side conditions of the subderivation on the righthand side, the following CHCs are obtained:

$$p_3() \Leftarrow p_1(h) \wedge h = 0,$$
$$p_2(h,i) \Leftarrow p_1(h) \wedge h = 0 \wedge i = 0.$$

CHCs are also obtained from the other subderivation in a similar manner. □

5 Implementation and Experiments

This section reports an implementation and experimental results.

5.1 Implementation

We have implemented a prototype program verifier for a subset of OCaml, which supports first-order functions, integers, and recursive data structures, based on the type inference procedure described above. As the backend CHC solvers, we employed multiple solvers: Z3 [12] ver. 4.8.12, HoIce [2] ver. 1.9.0, and Eldarica [4] ver. 2.0.7; that is because these solvers have pros and cons, and their running times vary depending on problem instances, as we report in Sect. 5.2.

As for the function F' in Sect. 4.1, the current implementation supports only the following functions $f_{i,\diamond} \in \mathtt{int}^{\ell_k+m_k} \rightarrow \mathtt{int}$ with $i \in \{1,2,3\}$ and $\diamond \in \{+, \max, \min\}$ (where n' in Sect. 4.1 is set to 1).

$$\begin{aligned}
f_{1,\diamond}(x_1,\dots,x_{\ell_k},y_1,\dots,y_{m_k}) &= \begin{cases} 1+(y_1 \diamond \cdots \diamond y_{m_k}) & \text{if } m_k > 0 \\ 0 & \text{otherwise} \end{cases} \\
f_{2,\diamond}(x_1,\dots,x_{\ell_k},y_1,\dots,y_{m_k}) &= x_1 \diamond \cdots \diamond x_{\ell_k} \diamond y_1 \diamond \cdots \diamond y_{m_k} \\
f_{3,\diamond}(x_1,\dots,x_{\ell_k},y_1,\dots,y_{m_k}) &= x_1 \diamond \cdots \diamond x_{\ell_k},
\end{aligned}$$

and chooses $f_{1,+}, f_{2,+}, f_{3,+}, f_{1,\max}, f_{2,\max}, f_{3,\max}, f_{1,\min}, f_{2,\min}, f_{3,\min}$ in this order, at each iteration. (Here, max and min are operations over integers extended with $-\infty$ and ∞.) In the case of lists, $f_{1,+}$, $f_{2,+}$, $f_{3,+}$, $f_{2,\max}$, and $f_{2,\min}$ can be used for computing the length, the sum of elements, the head element, the maximal element, and the minimal element of a list, respectively; since $f_{1,\max}$ and $f_{1,\min}$ coincide with $f_{1,+}$ for lists, it will be excluded out. Since the set of functions added as F' is finite, the current implementation obviously does not satisfy the relative completeness discussed in Sect. 4.1. Supporting more functions is not difficult in theory, but because the current implementation seemed to have already hit a certain limitation of the state-of-the-art CHC solvers (as reported in the next subsection), we plan to add more functions only after more efficient CHC solvers become available.

5.2 Experiments and Results

To evaluate the effectiveness of our approach, we have tested our prototype tool for several list/tree-processing programs. The experiments were conducted on a machine with Ubuntu 20.04.1 on Windows Subsystem for Linux 2, AMD Ryzen 7 3700X 8-Core Processor, and 16GB RAM.

Table 1. The experimental results

Program	#Lines	n	Time [s]	CHC solver	#clauses	#pvars
List-sum	17	2	2.25	HoIce	25	12
List-max	20	2	2.33	Z3	26	12
List-sorted	19	3	3.08	Z3	46	22
Range-basic	12	1	1.52	HoIce	16	8
Range-len (Ex. 1)	15	1	1.81	HoIce	25	12
Range-concat-len	21	1	2.61	HoIce	58	22
Isort-len	28	1	2.39	HoIce	66	27
Isort-is-sorted (Ex. 2)	30	3	4.32	Z3	79	33
Msort-len	45	—	—	—	145	49
Msort-is-sorted	52	—	—	—	161	54
Tree-size (Ex. 3)	15	1	1.95	HoIce	32	14
Tree-depth	21	1	2.07	HoIce	34	15
Bst-size	20	1	2.65	HoIce	64	28
Bst-sorted	51	—	—	—	148	74

Table 1 summarizes the experimental results. The benchmark set consists of the following programs.

- "list-sum" takes an integer m as an input, randomly generates a list so that the sum of elements is m, and then checks that the sum of elements is indeed m. Similarly, "list-max" generates a list so that the maximum element is m, and checks that the maximum element is indeed m, and "list-sorted" randomly generates a sorted list and checks that the list is indeed sorted.
- "range-X" generates a list $[m; m-1; \cdots ; 1]$ using the function `range` in Example 1, and checks its properties, where the property is "$n = 0$ if the generated list is null, and $m > 0$ otherwise" for X=basic, "the length is m" for X=len. The program "range-concat-len" calls `gen`(m) twice, concatenates the two lists, and check that the length of the resulting list is $2m$.
- "isort-X" takes an integer m as an input, generates a list of length m, sorts it with `isort` in Example 2, and checks properties of the resulting list, where the property is "the length of the list is m" for X=len, and "the list is sorted" for X=sorted.
- "msort-X" is a variation of "isort-X", where `isort` is replaced with a function for the merge sort.
- "tree-size" ("tree-depth", resp.) takes an integer m as an input, generates a tree of size (depth, resp.) m, and checks that the size (depth, resp.) of the tree is indeed m (for X=size).
- "bst-X" generates a binary search tree of a given size, and checks that the tree has the expected size (for X=size) or that the tree is a valid binary search tree (for X=sorted).

Appendix A shows some of the concrete programs used in the experiments.

In the table, the column "#Lines" shows the number of lines of the program (excluding empty and comment lines), and the column "n" shows the final value of n in Fig. 4, when the verification succeeded; the cell filled with "—" indicates a timeout (due to the backend CHC solver), where the time limit was set to 300 s. The columns "Time" and "CHC solver" show the running time and the backend CHC solver. Actually, we have run our tool for each of the three CHC solvers: Z3 [12] ver. 4.8.12, HoIce [2] ver. 1.9.0, and Eldarica [4] ver. 2.0.7, and the table shows only the best result. The result for other solvers are reported in Appendix A. The columns "#clauses" and "#pvars" show the numbers of output clauses and predicate variables, respectively (which do not depend on the value of n).

The results show that our tool works reasonably well: we are not aware of *fully automated* tools that can verify most of those programs. Our tool failed, however, to verify "msort-len", "msort-is-sorted", and "bst-sorted". To analyze the reason, we have manually prepared an optimal choice of functions $\widetilde{F}$ for those problems, and run the CHC solvers for the resulting CHC problems. None of the CHC solvers could solve the problems in time. This indicates that the main bottleneck in the current tool is not the choice of functions $\widetilde{F}$ discussed in Sects. 4.1 and 5.1, but rather the backend CHC solver. We expect that "msort-len", "msort-is-sorted", and "bst-sorted" can be automatically verified by our method if a more efficient CHC solver becomes available. It would be, however, important also to improve the heuristics for choosing n and $\widetilde{F}$, as briefly discussed in Remark 3.

6 Related Work

As already mentioned in Sect. 1, the idea of parameterizing recursive types with indices to represent various properties goes back at least to Xi and Pfenning's work on dependent ML [23,24]. In their system, however, explicit declarations of refinement types are required for data constructors and recursive functions. Kawaguchi et al. [5] introduced recursive refinement types, which allows a restricted form of parameterization of datatypes with predicates, and Vazou et al. [21] have introduced abstract refinement types, which are refinement types parameterized with predicates. Like Xi and Pfenning's system (and unlike ours), those systems also require explicit declarations of abstract refinement types for datatype constructors and/or functions, although refinement parameters in the code part can be omitted and automatically inferred (cf. [21], Sect. 3.4). The type system of Vazou et al. [21] supports polymorphism on predicates, unlike our type system.

The reduction from (ordinary) refinement types to the CHC satisfiability problem has been well studied [2,3,19]; we used that technique in Step 3 of our type inference procedure. The problem of inferring parameterized recursive refinement types appears to be related with that of inferring implicit parameters in refinement type systems [17,20]. In fact, Tondwalkar et al. [17] reduced the inference problem to the problem of solving existential CHCs, an extension of the CHC problem, and our problem of inferring P and F can also be reduced to that problem. We, however decided not to take that approach, because efficient solvers for existential CHCs are not available.[4] We instead designed a heuristic procedure to construct F, and reduced the rest of the inference problem to the satisfiability problem for ordinary CHCs.

There have been other (non-type-based) approaches to verification of programs manipulating recursive data structures. The series of work on TVLA [10, 11] targets programs with destructive updates, and infers the shape of data structures by using a 3-valued logic. Besides the difference in the target programs, to our knowledge, their analysis fixes predicates used for abstraction a priori (e.g., in [10], "instrumentation predicates" are specified by a user of the tool), whereas our tool fixes only the set of functions F_j's for mapping data structures to integers, and leaves it to the underlying CHC solver to find appropriate predicates. Thanks to the type-based approach, our approach can also be naturally extended to deal with higher-order programs.

7 Conclusion

We have introduced *parameterized recursive refinement types* (PRRT) that can express various properties of recursive data structures in a uniform manner, and proposed a type inference procedure for PRRT, to enable fully automatic verification of functional programs that use recursive data structures. We have implemented a prototype automated verification tool, and confirmed that the tool can automatically verify small but non-trivial programs. Future work includes an extension of the verification tool for a full-scale functional language, and a further refinement of the type inference procedure to improve the efficiency of the tool.

Acknowledgments. We would like to thank anonymous referees for useful comments. This work was supported by JSPS KAKENHI Grant Number JP20H05703.

Appendix

A Details of Experiments

Table 2 presents the experimental results for each backend CHC solver. The columns "n" and "Time" for each solver have the same meaning as Sect. 5.2.

[4] The work of Tondwalkar et al. [17] does not suffer from this problem, since the existential CHCs obtained in their work is acyclic, while the existential CHCs generated from our inference problem would be cyclic.

Table 2. The results of verification with three solvers

	Z3		HoIce		Eldarica	
Program	n	Time [s]	n	Time [s]	n	Time [s]
List-sum	2	2.34	2	2.25	2	4.26
List-max	2	2.33	2	2.39	2	6.79
List-sorted	3	3.08	—	—	3	23.07
Range-basic	1	1.60	1	1.52	1	1.80
Range-len	—	—	1	1.81	1	2.92
Range-concat-len	—	—	1	2.61	—	—
Isort-len	—	—	1	2.39	—	—
Isort-is-sorted	3	4.32	—	—	—	—
Msort-len	—	—	—	—	—	—
Msort-is-sorted	—	—	—	—	—	—
Tree-size	—	—	1	1.95	1	8.73
Tree-depth	—	—	1	2.07	—	—
Bst-size	—	—	1	2.65	—	—
Bst-sorted	—	—	—	—	—	—

As examples of the benchmark programs, Listings 18.1 and 18.2 respectively show the programs named "list-sum" and "bst-size" in Sect. 5.2.

Listing 18.1. Program list-sum

```
type list = Nil | Cons of int * list

let rec gen n =
  if n = 0 then Nil
  else
    let x = Random.int (n + 1) in
    Cons(x, gen (n - x))

let rec sum xs =
  match xs with
  | Nil -> 0
  | Cons(x, xs) -> x + sum xs

let rec main n =
  if n >= 0 then
    let s = sum (gen n) in
    assert (s = n)
  else
    0
```

Listing 18.2. Program bst-size

```
type bst = Leaf | Node of int * bst * bst

let rec insert t x =
  match t with
    | Leaf           -> Node(x, Leaf, Leaf)
    | Node(y, l, r) ->
      if x < y then Node(y, insert l x, r)
      else Node(y, l, insert r x)

let rec gen n =
  if n = 0 then Leaf
  else insert (gen (n - 1)) (Random.int 10000)

let rec size t =
  match t with
    | Leaf           -> 0
    | Node(_, l, r) -> 1 + size l + size r

let rec main n =
  if n >= 0 then
    let g = size (gen n) in
    assert (g = n)
  else
    0
```

References

1. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
2. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: ICE-based refinement type discovery for higher-order functional programs. J. Autom. Reason. **64**(7), 1393–1418 (2020). https://doi.org/10.1007/s10817-020-09571-y
3. Hashimoto, K., Unno, H.: Refinement type inference via horn constraint optimization. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 199–216. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48288-9_12
4. Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–7 (2018)
5. Kawaguchi, M., Rondon, P.M., Jhala, R.: Type-based data structure verification. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15–21, 2009, pp. 304–315. ACM (2009). https://doi.org/10.1145/1542476.1542510
6. Kobayashi, N., Sato, R., Unno, H.: Predicate abstraction and CEGAR for higher-order model checking. In: PLDI 2011, pp. 222–233. ACM Press (2011)

7. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods Syst. Design **48**(3), 175–205 (2016). https://doi.org/10.1007/s10703-016-0249-4
8. Kuwahara, T., Terauchi, T., Unno, H., Kobayashi, N.: Automatic termination verification for higher-order functional programs. In: Shao, Z. (ed.) ESOP 2014. LNCS, vol. 8410, pp. 392–411. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54833-8_21
9. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Hankin, C., Schmidt, D. (eds.) Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17–19, 2001, pp. 81–92. ACM (2001). https://doi.org/10.1145/360204.360210
10. Lev-Ami, T., Sagiv, M.: TVLA: a system for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–301. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-45099-3_15
11. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, M.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 181–198. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_13
12. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
13. Mukai, R., Kobayashi, N., Sato, R.: Parameterized recursive refinement types for automated program verification (2022), a longer version of this paper, available from http://www.kb.is.s.u-tokyo.ac.jp/~koba/
14. Ong, C.H.L., Ramsay, S.: Verifying higher-order programs with pattern-matching algebraic data types. In: Proceedings of POPL, pp. 587–598. ACM Press (2011)
15. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI 2008, pp. 159–169 (2008)
16. Sato, R., Unno, H., Kobayashi, N.: Towards a scalable software model checker for higher-order programs. In: Proceedings of PEPM 2013, pp. 53–62. ACM Press (2013)
17. Tondwalkar, A., Kolosick, M., Jhala, R.: Refinements of futures past: Higher-order specification with implicit refinement types. In: Møller, A., Sridharan, M. (eds.) 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11–17, 2021, Aarhus, Denmark (Virtual Conference). LIPIcs, vol. 194, pp. 18:1–18:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ECOOP.2021.18
18. Unno, H., Kobayashi, N.: On-Demand refinement of dependent types. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 81–96. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78969-7_8
19. 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, September 7–9, 2009, Coimbra, Portugal, pp. 277–288. ACM (2009)
20. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2013, pp. 75–86. ACM (2013)

21. Vazou, N., Rondon, P.M., Jhala, R.: Abstract refinement types. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 209–228. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_13
22. 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, Gothenburg, Sweden, September 1–3, 2014, pp. 269–282. ACM (2014)
23. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: Davidson, J.W., Cooper, K.D., Berman, A.M. (eds.) Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17–19, 1998, pp. 249–257. ACM (1998). https://doi.org/10.1145/277650.277732
24. Xi, H., Pfenning, F.: Dependent types in practical programming. In: Proceedings of POPL, pp. 214–227 (1999)
25. Zhu, H., Nori, A.V., Jagannathan, S.: Learning refinement types. In: Proceedings of ICFP 2015, pp. 400–411. ACM (2015). https://doi.org/10.1145/2784731.2784766