Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Verifying relational properties of functional programs by first-order refinement ☆

Kazuyuki Asada *, Ryosuke Sato *, Naoki Kobayashi *

University of Tokyo, Japan

ARTICLE INFO

Article history:
Received 6 July 2015
Received in revised form 25 February 2016
Accepted 27 February 2016
Available online xxxx

Keywords:
Automated verification
Higher-order functional language
Refinement types

ABSTRACT

Much progress has been made recently on fully automated verification of higher-order functional programs, based on refinement types and higher-order model checking. Most of those verification techniques are, however, based on *first-order* refinement types, hence unable to verify certain properties of functions (such as the equality of two recursive functions and the monotonicity of a function, which we call *relational properties*). To relax this limitation, we introduce a restricted form of higher-order refinement types where refinement predicates can refer to functions, and formalize a systematic program transformation to reduce type checking/inference for higher-order refinement types to that for first-order refinement types, so that the latter can be automatically solved by using an existing software model checker. We also prove the soundness of the transformation, and report on implementation and experiments.

1. Introduction

There has been much progress in automated verification techniques for higher-order functional programs [12,17,16,9,11,19,13].[1] Most of those techniques abstract programs by using *first-order* predicates on base values (such as integers), due to the limitation of underlying theorem provers and predicate discovery procedures. For example, consider the program:

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

Using the existing techniques [12,17,16,9], one can verify that `sum` has the first-order refinement type: $(n : \mathbf{int}) \to \{m : \mathbf{int} \mid m \geq n\}$, which means that `sum n` returns a value no less than `n`. Here, $\{m : \mathbf{int} \mid P(m)\}$ is the (refinement) type of integers m that satisfy $P(m)$.

Due to the restriction to the first-order predicates, however, it is difficult to reason about what we call *relational properties*, such as the relationship between two functions, and the relationship between two invocations of a function. For example, consider another version of the sum function:

☆ This article is a revised and extended version of the paper that appeared in Proceedings of PEPM 2015 under the title "Verifying Relational Properties of Functional Programs by First-Order Refinement". We have added detailed definitions, proofs, and explanations.

* Corresponding authors.
E-mail addresses: asada@kb.is.s.u-tokyo.ac.jp (K. Asada), ryosuke@kb.is.s.u-tokyo.ac.jp (R. Sato), koba@kb.is.s.u-tokyo.ac.jp (N. Kobayashi).

[1] In the present paper, by *automated* verification, we mean (almost) fully automated one, where a tool can automatically verify a given program satisfies a given specification (expressed either in the form of assertions or refinement type declarations), without requiring invariant annotations (such as pre/post conditions for each function). It should be contrasted with refinement type checkers [20,2] where a user must declare refinement types for *all* recursive functions including auxiliary functions. Some of the automated verification techniques above require a hint [19], however.

http://dx.doi.org/10.1016/j.scico.2016.02.007
0167-6423/

```
let rec sumacc n m = if n<0 then m else sumacc (n-1) (m+n)
and sum2 n = sumacc n 0
```

Suppose we wish to check that $\texttt{sum2}(n)$ equals $\texttt{sum}(n)$ for every integer n. With general refinement types [6], that would amount to checking that `sumacc` and `sum2` have the following types[2]:

$$\texttt{sumacc} : (n : \mathbf{int}) \to (m : \mathbf{int}) \to \{r : \mathbf{int} \mid r = m + \texttt{sum}(n)\}$$
$$\texttt{sum2} : (n : \mathbf{int}) \to \{r : \mathbf{int} \mid r = \texttt{sum}(n)\}$$

The type of `sum2` means that `sum2` takes an integer as an argument n and returns an integer r that equals the value of $\texttt{sum}(n)$. With the first-order refinement types, however, `sum` cannot be used in predicates, so the only way to prove that $\texttt{sum2}(n)$ equals $\texttt{sum}(n)$ would be to verify precise input/output behaviors of the functions:

$$\texttt{sum}, \texttt{sum2} : (n : \mathbf{int}) \to \{r : \mathbf{int} \mid (n \geq 0 \wedge r = n(n+1)/2) \vee (n < 0 \wedge r = 0)\}.$$

Since this involves non-linear and disjunctive predicates, automated verification (which involves automated synthesis of the predicates above) is difficult. In fact, most of the recent automated verification tools do not deal with non-linear arithmetic.

Actually, with the first-order refinement types, there is a difficulty even with the "trivial" property that `sum` satisfies $\texttt{sum}\ x = x + \texttt{sum}\ (x-1)$ for every $x \geq 0$. This is almost the definition of the `sum` function, and it can be expressed and verified using the general refinement type:

$$\texttt{sum} : \{f : \mathbf{int} \to \mathbf{int} \mid \forall x.\ x \geq 0 \Rightarrow f(x) = x + f(x-1)\}.$$

Yet, with the restriction to first-order refinement types, one would need to infer the precise input/output behavior of `sum` (i.e., that $\texttt{sum}(x)$ returns $x(x+1)/2$).[3]

We face even more difficulties when dealing with higher-order functions. Consider the following program.

```
let nil i = None in
let tl xs = fun i-> xs(i+1) in
let cons x xs = fun i -> if i=0 then Some(x) else xs(i-1) in
let rec append xs ys =
  match xs(0) with None -> ys
                 | Some(x) -> let xs' = tl xs in cons x (append xs' ys)
```

Here, a list is encoded as a function that maps each index to the corresponding element (or `None` if the index is out of bounds) [13], and the append function is defined. Suppose that we wish to verify that `append xs nil = xs`. With general refinement types, the property would be expressed by:

$$\texttt{append} : (x : \mathbf{int} \to \mathbf{int\ option}) \to \{y : \mathbf{int} \to \mathbf{int\ option} \mid y(0) = \texttt{None}\} \to \{r : \mathbf{int} \to \mathbf{int\ option} \mid r = x\}$$

(where $r = x$ means the extensional equality of functions r and x) but one cannot directly express and verify the same property using first-order refinement types.

To overcome the problems above, we allow[4] programmers to specify (a restricted form of) general refinement types in source programs. For example, they can declare

$$\texttt{sum2} : (n : \mathbf{int}) \to \{r : \mathbf{int} \mid r = \texttt{sum}(n)\}$$
$$\texttt{append} : (x : \mathbf{int} \to \mathbf{int\ option}) \to (\{y : \mathbf{int} \to \mathbf{int\ option} \mid y(0) = \texttt{None}\} \to \{r : \mathbf{int} \to \mathbf{int\ option} \mid \forall i.r(i) = x(i)\}.$$

To take advantage of the recent advance of verification techniques based on first-order refinement types, however, we employ automated program transformation, so that the resulting program can be verified by using only first-order refinement types. The key idea of the transformation is to apply a kind of tupling transformation [3] to capture the relationship between two (or more) function calls at the level of first-order refinement. For example, for the `sum` program above, one can apply the standard tupling transformation (to combine two functions `sum` and `sumacc` into one) and obtain:

```
let rec sum_sumacc (n, m) =
  if n<0 then (0,m)
         else let (r1,r2)=sum_sumacc (n-1, m+n) in (r1+n, r2)
```

[2] As defined later, a formula $t_1 = t_2$ in a refinement type means that *if both t_1 and t_2 evaluate to (base) values*, then the values are equivalent.

[3] Another way would be to use uninterpreted function symbols, but for that purpose, one would first need to check that `sum` is total.

[4] But programmers are not obliged to specify types for all functions. In fact, for the example of `sum2`, no declaration is required for the function `sum`.

Please cite this article in press as: K. Asada et al., Verifying relational properties of functional programs by first-order refinement, Sci. Comput. Program. (2016), http://dx.doi.org/10.1016/j.scico.2016.02.007