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

Kodai Hashimoto, Sho Torii and Hiroshi Unno (University of Tsukuba)

{kodai, sho, uhiro}@logic.cs.tsukuba.ac.jp

Overview

We propose a refinement type based total correctness (i.e., **partial correctness** and **termination**) verification method of ML-like higher-order functional programs with ADTs.

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

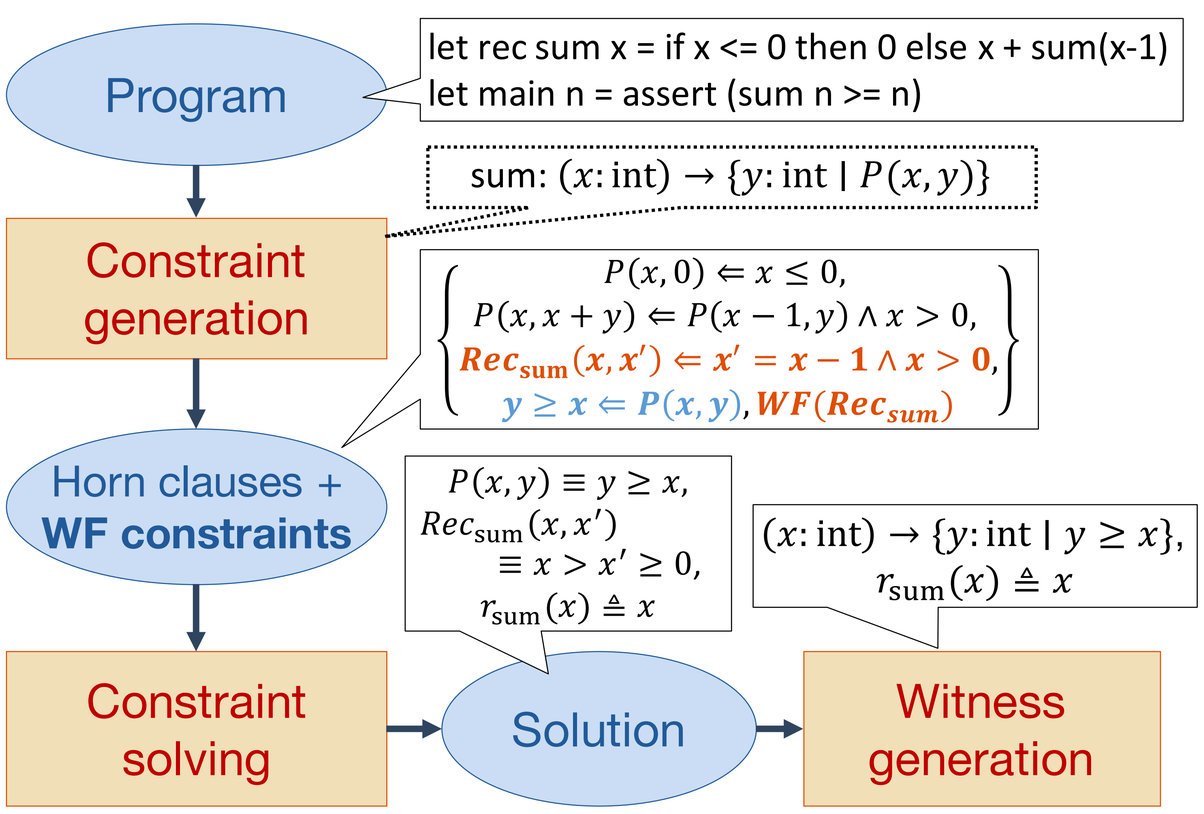
Witness of partial correctness → a refinement type: $(x\colon \mathrm{int}) \to \{y\colon \mathrm{int} \mid \boldsymbol{y \geq x}\}$

Our method

Witness of termination → a ranking function: $r_{\mathrm{sum}}(x) \triangleq x$

r_{sum} witnesses the well-foundedness of sum's **recursion relation** $\boldsymbol{Rec_{sum}} \triangleq \{(\boldsymbol{x}, \boldsymbol{x-1}) \mid \boldsymbol{x > 0}\}$, which represents relational invariant between the arguments **x** and **x − 1** of sum, because $Rec_{\mathrm{sum}} \subseteq \{(x, x') \mid r_{\mathrm{sum}}(x) > r_{\mathrm{sum}}(x') \geq 0\}$, and consequently the termination of sum is concluded (∵ there is no infinite descending chain in Rec_{sum})

Overall structure



How to handle functions on ADTs

Abstract ADTs into integers by using size functions.

Example

```
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
```

Our tool basically uses **syntactic sizes** of the data structures as size functions, but our tool can automatically synthesize other size functions if necessary.

Our method

a size function:

$$size([\,]) = 1$$
$$size(h :: t) = 1 + size(t)$$

a ranking function: $r_{\mathrm{merge}}(cmp, l1, l2) \triangleq size(l1) + size(l2)$

How to handle higher-order functions

Introduce an ADT that encodes the closures possibly passed as function arguments, and insert a ghost parameter of the ADT just before each function argument.

Example

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

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

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

Our method

a size function:

$$size(G_{rec}(x)) = x$$
$$size(G) = size(App(cls)) = 0$$

a ranking function: $r_{\mathrm{g}}(x) \triangleq x$

Implementation and Experiments

We have implemented a **fully-automated type checking and inference tool** called *Refinement Caml.*

Successfully verified OCaml's List and Map modules:

Mod	LOC	Fun	Size	Inv	Rank	Time
List	395	44	0	2	0	22.5
Map	305	23	0	0	0	39.4

Successfully verified tricky recursive functions:

- Ackermann
- McCarty 91
- quicksort
- Examples from
  - higher-order binary reachability analysis [Kuwahara et al. '14]
  - higher-order size-change analysis [Sereni '06, '07] [Jones et al. '08]

Refinement Caml offers many other features for solving a wide variety of verification problems that cannot be described here due to space constraints.

Refinement Caml supports:

- size function inference
- lexicographic linear ranking functions synthesis
- (maximally-)weak precondition inference
- conditional termination verification
- program games solving
- etc…

Try Refinement Caml