On Generating Constraints for Refinement Type Inference

Ruslán Ledesma-Garza and Andrey Rybalchenko

Technische Universität München

Length of talk: 30 minutes

Refinement types is a viable approach to verification of higher order functional programs. The key idea is to use the program source code to generate implication constraints over unknown, to be discovered refinement predicates and then solve these constraints using abstraction, refinement, interpolation and other SMT based techniques, e.g., consider the tools DSolve, DepCegar, MoCHi. In this paper we revisit how constraints on refinement predicates are generated in order to identify and explore potential scalability improvements for refinement type inference. We present a constraint generation algorithm that only keeps track of refinement predicates for calling contexts and function types of procedures, thus avoiding generation of (constraints over) refinement predicates for intermediate sub-expressions in procedure bodies. Furthermore, we perform uncurrying of procedures whenever possible, to avoid generation of refinement predicates for each formal argument, while still supporting partial applications. Our implementation of the algorithm shows that order of magnitude reductions of the number of refinement predicates and sub-typing constraints can be achieved in practice, which subsequently leads to great improvements in solving time.

As an illustration of our approach we present a simple example below.

```
let rec add x y = if x = 0 then y else 1 + add (x - 1) y
let main m n    = assert (add m n = m + n)
let _           = main (read_int ()) (read_int ())
```

Using a state-of-the-art constraint generation algorithm, e.g., DSolve, we obtain 34 constraints over 18 refinement predicates. For example, the function `add` will have a refinement type

$$x : \{\text{int} \mid A\} \to y : \{\text{int} \mid B\} \to \{\text{int} \mid C\}$$

where A, B, and C are refinement predicates to be discovered. In `add`, each sub-expression will be decorated with a refinement predicate, e.g., $V = y + x$. In contrast, our algorithm generates the following 6 constraints over 6 refinement predicates.

$$\begin{array}{ll}
add_{context}(X-1,Y) & \leftarrow X \neq 0 \wedge add_{context}(X,Y) \\
 & \quad \text{parameter passing for recursive application of } \texttt{add} \\
add_{summary}(X,Y,R) & \leftarrow (X = 0 \wedge R = Y \vee X \neq 0 \wedge add_{summary}(X-1,Y,R-1)) \wedge add_{context}(X,Y) \\
 & \quad \text{summary of } \texttt{add} \\
add_{context}(M,N) & \leftarrow main_{context}(M,N) \\
 & \quad \text{parameter passing for application of } \texttt{add} \text{ in } \texttt{main} \\
R = M + N & \leftarrow add_{summary}(M,N,R) \wedge main_{context}(M,N) \\
 & \quad \text{assertion in } \texttt{main} \\
main_{summary}(M,N,_) & \leftarrow R = M + N \wedge add_{summary}(M,N,R) \wedge main_{context}(M,N) \\
 & \quad \text{summary of } \texttt{main} \\
main_{context}(_,_) & \leftarrow true \\
 & \quad \text{parameter passing for application of } \texttt{main}
\end{array}$$

A solution to our set of constraints can be translated to a solution for constraints generated by DSolve by applying an equivalent of the intra-procedural strongest post-condition computation. That is, our constraints can be seen as a result of applying existential quantifier elimination on the DSolve constraints, where some of the existentially quantified refinement predicates are eliminated. Since such an elimination is difficult to implement once the constraints are generated, our constraint generation algorithm avoids introduction of such refinement predicates in the first place.