Hype for Refinements

Hype for Types: Lecture 8
Jeanne Luning Prak

March 2018

1 Motivation

Consider a function

$$\texttt{f} : \texttt{int} \to \texttt{int} \to \texttt{int}$$

What does this function do? We know from the type that it takes in two integers and returns an integer, but there are many such functions that do this. Is it multiplication? Addition? Does it just return its second argument?

There's a mantra among programming language theorists: well-typed programs don't go wrong. The type should determine the behavior of the program. And to some extent, it does: we know that an `int` will never magically turn itself into a string while our program is running, and we know that `f` can always be applied to two integers. However, if I want to know if `f` is multiplication or addition, I'm out of luck. In particular, if I'm a programmer trying to implement `f`, and I want to implement multiplication, it's up to me to get it right. My type system won't stop me if I try to implement addition instead.

It seems rather silly that I would be worried about implementing the wrong arithmetic operation, but imagine instead I'm implementing a function

$$\texttt{printSSN} : \texttt{user} \to \texttt{string option}$$

which returns a string of sensitive information (a social security number) if the user is authorized, and `NONE` otherwise. The consequences of me writing this function incorrectly are huge. However, again, the type system does not help me.

These examples motivate our discussion of *refinements*, logical predicates attached to types that specify the behavior of an expression. As we shall see, refinements allow us to express very powerful data invariants, preconditions, and postconditions. However, they can often be cumbersome to work with, and it is an active area of research to make them more usable.

2 Refinements

A *refinement* is a logical predicate attached to a type that describes the behavior of an expression. The refinement and type

$$e : \tau\{\phi\}$$

describes that $e : \tau$ and also has some predicate ϕ that holds about it. The refinement ϕ is attached to τ. If e does not have type τ, then ϕ is not even checked; it automatically doesn't hold. For this reason, we sometimes refer to types with refinements as "refinement types" since refinements cannot be separated from the types they refine.

For example,

$$e : \texttt{int}\{\texttt{greaterThanZero}\}$$

describes an expression e, of type int, with the refinement `greaterThanZero`, which states that when e evaluates to a value, e must be greater than 0. Similarly, we could also have

$$e : \texttt{int} \to \texttt{int}\{\texttt{safe}\}$$

which describes an expression e that has type $\texttt{int} \to \texttt{int}$, and does not raise a runtime error like division by zero.

3 Dependent Types

Additionally, we'd like to be able to have refinements depend on terms in our program. Maybe I have a tuple

$$(x, y)$$

and I want to express $x < y$ in a refinement. I can use this by introducing refinements that depend on program terms. These are referred to as dependent refinement types. We modify our syntax to have a variable referring to the program term:

$$(\texttt{x}, \texttt{y}) : (\texttt{int} * \texttt{int})\{\texttt{v} \mid \texttt{v.l} < \texttt{v.r}\}$$

where v refers to the value we're refining (in this case, (x, y)) and `.l` and `.r` refer to the left element of the tuple and the right element of the tuple respectively. We can use this to refine functions as well

```
fun mult (x : int, y : int) : int{r | r = x * y} =
    if y = 0 then 0 else mult (x, y - 1) + x
```

Using this, we can create something very simliar to contracts in 15-122, but enforced by our type system!

4 Static vs. Dynamically Checked Programs

In SML, programs are typechecked statically. That is, you can always tell the type of an expression without running that expression. However, in a language with refinements, it may not be possible to statically check the refinements. This is because checking most non-trivial properties about programs is *undecidable*: it's impossible to write an algorithm that will correctly check that that particular property holds.

To see why this is, consider the most famous undecidable problem: the halting problem. The halting problem is simply: design an algorithm such that given a function and an input, output "YES" if it halts and "NO" if it loops forever. It's not too difficult to prove that this is an impossible problem to solve. No such algorithm could exist! However, if we allow any old logical predicate to be expressed in refinements, we could easily write:

$$f : (\texttt{int} \to \texttt{int})\{\texttt{halts}\}$$

and then refinement checking would instantly become impossible.

To avoid this, there are a bunch of things we can do.

1. Dynamically check our programs. This is what we do in 15-122. Contracts are checked at runtime, not at compile time. This allows us to check many more refinements, but it also limits the usefulness of refinements. Instead of guaranteeing that if our code compiles, it works, it guarantees that our code will crash rather than violate a refinement. This is still good! If `printSSN` crashes rather than printing a SSN to someone who shouldn't see it, it's better than the alternative. But it's less desirable than guaranteeing that our functions work at compile time.
2. Accept our limitations. We simply limit ourselves to refinements that are decidable. We can't verify as many things, but we can still verify interesting properties. This is the premise behind a very expressive refinement checker called LiquidHaskell[1].
3. Require the programmer to write proofs. We simply have our program refuse to typecheck unless the programmer proves all the refinements that an algorithm can't. This is used in the language F* to prove very powerful properties, like totality of functions[2].

[1] https://ucsd-progsys.github.io/liquidhaskell-blog/
[2] https://www.fstar-lang.org/